M Ű E G Y E T E M 1 7 8 2

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

Extensions and Generalization of the Saturation Algorithm in Model Checking

Ph.D. Dissertation

Vince Molnár

Thesis supervisor:
István Majzik, Ph.D. (BME)

Budapest
2019

Vince Molnár
http://mit.bme.hu/~molnarv/

June 2019

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.

doi: 10.5281/zenodo.TODO

Declaration of own work and references

I, Vince Molnár, 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 Molnár Vince 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, 2019. 09. 10.

Molnár Vince

Acknowledgements

Even though this dissertation is by far not an end of a road, it concludes a long journey that started 8 years ago. Along this journey, many people supported me in growing from an enthusiastic BSc student to an even more enthusiastic teacher and researcher.

First and foremost, I would like to thank my PhD advisor, Dr. István Majzik, as well as my BSc and MSc supervisors, Dr. András Vörös and Dr. Tamás Bartha. You are the people most directly involved in my professional development.

In this regard, I also want to thank the support and helpful advices of Prof. Dániel Varró, who motivated me to follow my path, Prof. András Pataricza, who provided endless opportunities to test myself and improve, and Dr. Zoltán Micskei, who gently steered me in managing myself.

The quality of the final version of this dissertation is also the result of the helpful comments and recommendations of my reviewers, Prof. Zoltán Fülöp and Prof. Jaco van de Pol. Thank you again for your valuable work!

I am extremely grateful for the persistent support of my family, who helped me with so many aspects of my life. Without you, I would not be where I am today.

Eszti, Zsófi, you were with me all this time, providing support when I needed it the most. You endured even the toughest times when I didn't have the time and energy to return your kindness. Your part in this work is invisible to the readers, but it was in fact essential.

I am also grateful to all my colleagues, who became my friends during this time and form a group I am truly happy to be part of. To my students, who often helped my research goals, and inspired me with their talent. Gábor, Ati, Ákos, Dávid, Dani, Marci, Kristóf, Bence, Levi, and the many of you not mentioned by name, I hope we will often cross each other's paths one way or another!

To my fellow researchers in the world, especially Jeroen and Yann, thank you for the fruitful discussions and the nice time we spent together in different parts of the continent, I hope we will work together in the future!

Last but not least, I am grateful for the financial support provided by the MTA-BME Lendület Cyber-Physical Systems Research Group, the New National Excellency Program and the National Talent Program of the Ministry of Human Capacities.

Summary

Formal verification is a collective name of techniques that aim to *prove* that a system design or implementation satisfies its required properties. Formally proving something is much more challenging than testing and simulation: the proof has to cover every possible case, not just selected executions of the system. This kind of rigorous analysis is often used in the development of safety-critical hardware and software systems, commonly found in the automotive, railway or aerospace industries.

Model checking is an automated formal verification technique that evaluates the possible states and behaviors of a system design with regard to a formal specification. This is a challenging task because real-life systems usually have an enormous number of possible states, all (or most) of which has to be checked for violations of the specified properties. Symbolic model checking techniques, and in particular the saturation algorithm have been introduced to efficiently handle these large state spaces in many (common) cases.

Research of the saturation algorithm has yielded different variants suitable for different problems, but the range of possible applications are still not fully covered. This dissertation presents further results in the fields of efficient analysis of models with prioritized transitions and the model checking of linear temporal logic properties, a class of specification logic that is prevalent but hard to analyze. Furthermore, investigation of different variants of saturation inspired an enhanced and more general version of the original algorithm, which is the culmination of this research.

Thesis 1 presents an approach to efficiently analyze Generalized Stochastic Petri Nets, a formalism suitable for the modeling of extra-functional aspects of a system. These models have prioritized transitions, which is challenging for saturation. The presented approach decomposes the priority-related aspects from the effect of the transition and achieves better scaling compared to previous approaches, expanding the scope of models on which analysis is feasible.

Thesis 2 is about the model checking of linear temporal logic properties, which is a complex algorithm only partially supported by saturation. It investigates how saturation can be used in different phases of the algorithm and introduce extensions accordingly. Furthermore, Thesis 2 introduces new algorithms for the different subtasks of this model checking problem, tailored to work with saturation and combining the best of previously known approaches. The combination of these building blocks provides a scalable, optimized symbolic model checking algorithm for linear temporal logic properties.

Thesis 3 introduces an enhanced version of the saturation algorithm. By identifying the common ideas, the enhanced version generalizes previous variants in a single, more powerful algorithm that can directly tackle a wider range of problems. Enhancement can be observed on performance as well: in certain types of models, the enhanced version outperforms the original saturation algorithm by orders of magnitude, while it has the same performance in other cases.

Összefoglaló

Formális verifikáció alatt olyan technikákat értünk, amelyek célja egy rendszerterv vagy -implementáció helyességének *bizonyítása* adott követelmények szempontjából. Egy követelmény teljesülésének bizonyítása sokkal nagyobb kihívás, mint a tesztelés vagy szimuláció, hiszen egy bizonyításnak az összes eshetőséget le kell fednie, nem csak (válogatott) egyedi lefutásokat. Ez a fajta szigorú analízis gyakran elvárás biztonságkritikus hardver- és szoftverrendszerek fejlesztésekor, amik gyakoriak pl. a jármű-, vasúti és repülőgépiparban.

A modellellenőrzés egy olyan automatikus formális verifikációs módszer, amely egy rendszerterv (modell) által leírt lehetséges állapotokat és viselkedéseket vizsgál a rendszer specifikációja szempontjából. Ez egy nehéz probléma, ugyanis a legtöbb valós rendszernek hatalmas számú lehetséges állapota van, amiket mind (de legalábbis nagy részét) meg kell vizsgálni, hogy nem sérti-e a specifikált rendszertulajdonságokat. Ezeknek a nagyméretű állapottereknek a hatékony kezelésére fejlesztették ki a szimbolikus modellellenőrzés módszerét, azon belül is a rendkívül hatékony ún. szaturáció algoritmust.

A szaturáció algoritmust többféle probléma hatékony megoldására is adaptálták, de a lehetséges alkalmazási területek jó része még lefedésre vár. Jelen disszertáció ebben az irányban mutat be új eredményeket, adaptálva az algoritmust prioritásos viselkedésmodellek, illetve lineáris idejű temporális logika segítségével megfogalmazott követelmények hatékony ellenőrzésére. lineáris idejű temporális logikai kifejezéseket széles körben alkalmaznak követelmények formális leírására, azonban ellenőrzésük nehéz feladat. Mindezeken felül, a disszertációban bemutatott kutatás csúcspontjaként a szaturáció különböző variánsainak elemzéséből inspirálódva továbbfejlesztésre került maga a szaturáció algoritmus is.

Az 1. tézis az ún. Általánosított Sztochasztikus Petri-hálók (Generalized Stochastic Petri Net) modellezési formalizmus hatékony elemzésére mutat be új, szaturáció alapú megközelítést. Ez a nyelv a rendszerek extrafunkcionális tulajdonságainak modellezésére szolgál, és a rendszer lehetséges viselkedései (állapotátmenetei) között prioritásokat is definiál, amelyek kezelése kihívás a szaturáció számára. A bemutatott megoldás szétválasztja az állapotátmenetek prioritással kapcsolatos leírását és az átmenet hatásának leírását, amivel jobb skálázódás érhető el a korábbi megoldásokhoz képest, ezzel növelve az elemezhető modellek körét.

A 2. tézis a lineáris idejű temporális logikai tulajdonságok ellenőrzéséről szól. Az erre szolgáló algoritmus több összetett lépésből áll, melyek közül a szaturáció nem mindent támogat. A kapcsolódó fejezetekben megvizsgálom, hogy milyen kiterjesztésekkel lehet a szaturációt az algoritmus minél több fázisában használni, valamint a szaturációhoz jól illeszkedő új algoritmusokat dolgozok ki és mutatok be, kombinálva a korábbi megoldások előnyeit. Mindezek az eredmények együttesen egy jól skálázódó, optimalizált szimbolikus modellellenőrző algoritmust adnak lineáris idejű temporális logikai tulajdonságok ellenőrzésére.

A 3. tézisben bemutatom a szaturáció algoritmus továbbfejlesztett változatát. A korábbi variánsok közös elemeinek elemzésével létrehoztam az algoritmus egy olyan általánosabb és hatékonyabb változatát, ami a korábbi variánsok kiváltása mellett egy, az eredeti algoritmushoz képest szélesebb problémakör megoldására is alkalmas. A továbbfejlesztés a teljesítményen is megfigyelhető: egyes modelltípusok esetén a továbbfejlesztett változat akár több nagyságrenddel is gyorsabb az eredetinél, míg más esetben is az eredetihez hasonló teljesítményt nyújt.

List of Abbreviations

Abbreviation	Introduced in	Description
$2k$-MDD	Section 2.3.4.3	Multi-valued Decision Diagram with $2k$ levels
ANSD	Definition 34	Abstract Next-State Descriptor
BDD	Section 2.3	Binary Decision Diagram
BFS	Section 2.3.5	Breadth-First Search
BNF	Section 2.1.4.2	Backus-Naur Form
CTL	Section 2.1.4.2	Computation-Tree Logic
DFS	Chapter 1	Depth-First Search
EVIDD	Definition 27	Edge-Valued Interval Decision Diagrams
EDD	Section 7.1	Edge-Valued Decision Diagram
GSA	Section 6.3	Generalized Saturation ALgorithm
GSPN	Section 2.1.2	Generalized Stochastic Petri Net
IDD	Section 7.1	Interval Decision Diagram
LTL	Section 2.1.4.2	Linear Temporal Logic
Hyb-MC	Section 5.4	Hybrid Model Checking (implementation of Thesis 2)
MCC	Section 5.4	Model Checking Contest
MDD	Definition 16	Multi-valued Decision Diagram
MxD	Definition 18	Matrix-Diagram
NS	Definition 34	Next-State [descriptor]
PTS	Definition 9	Partitioned Transition System
SA	Section 6.5.1	Saturation Algorithm
SCC	Section 2.2.1	Strongly Connected Component
SPN	Section 2.1.2	Stochastic Petri Net
SSG	Section 5.4.3.1	State Space Generation

Contents

Chapter 1

Introduction

Between 1985 and 1987, a software bug in the Therac-25 radiation therapy machine killed or severely injured at least six patients [LT93]. The problem was caused by improper synchronization between two concurrent processes, which lead to a race condition if the operator entered specific commands too fast. Race conditions are very hard to detect during testing because they appear only very rarely. They are a prime example of nondeterministic behavior, which is a mathematical abstraction of unpredictable but possible outcomes.

Incidents such as that of Therac-25 showed that software, especially concurrent software is very easy to get wrong, and the faults are very hard to detect with conventional testing approaches. By that time, formal methods in computer engineering had well-laid foundations in mathematical logic, which allowed engineers to specify, model and verify their designs in a mathematically precise way [BH14]. With a formal specification and model, the goal of formal verification is to *prove* the correctness of the modeled behavior in terms of the specification as opposed to testing certain executions. At the time of the Therac incidents, pioneering work in the automation of detecting concurrency problems has just been published by Edmund M. Clarke and Ernest A. Emerson [EC80] as well as Joseph Sifakis [QS82]: the technique of *model checking*.

A certain (abstract) behavior of a computer system can be represented by a sequence of *discrete states* that describe the system in a specific moment of time. The system would stay in a state for some amount of time, then *transition* into another one practically instantaneously. The whole behavior of the system can therefore be regarded as a graph (states as nodes and transitions as directed arcs), where multiple outgoing transitions denote nondeterministic choices (e. g. the scheduling of concurrent processes as described above). Every path, i. e. sequence of states in this graph is a possible execution of the system that might realize under some circumstances. Therefore, any potential error is also present somewhere in the graph, as an undesired state or sequence of states. The goal of model checking is to construct this *state graph* (also called *state space*) and look for the error as specified by some logic formula coming from the specification of system properties.

Obviously, the more complex the system, the more states it will have, and unfortunately the relationship is usually exponential – this is known as the *state space explosion problem*. In practice, a realistic system model would usually have so many states that it is impossible to store all of them in computer memory. One of the solutions proposed for this problem is *symbolic model checking* [Bur+92].

The main motivation of symbolic model checking is that states are usually vectors of values assumed by variables of the system, and many of these state vectors in the state space would be similar (especially in concurrent systems). For example, a thread in a program will most likely change only its local variables and some of the shared global variables, but will not affect local variables of any other thread. Symbolic model checking exploits this by characterizing *sets of state vectors* with Boolean functions (characteristic functions), which will describe the common features of and relations between the vectors in the set. For example, a function $f(x, y, z) = \neg x \vee y$ over Boolean variables x, y and z describe 6 vectors by returning *true* for exactly those triples that satisfy $\neg x \vee y$. Basic set operations needed for model checking then map to Boolean operators (union to disjunction, intersection to conjunction, complementation to negation).

As a compact representation for Boolean functions, Randal Bryant proposed binary *decision diagrams* [Bry86], which are essentially decision trees with all the identical subtrees merged. Manipulation of decision diagrams to execute logical operations can be very efficient with recursion and caching, as the merged subtrees have to be processed only once.

The recursive nature of decision diagrams has inspired a new model checking algorithm that – instead of using breadth-first search (BFS) or depth-first search (DFS) – follows the structure of the decision diagram to recursively compute the state graph through a series of submodels, each reused in the next one until the precise result is found. The algorithm suits decision diagrams very well and provides a fast and memory-efficient way to construct the decision diagram representing the set of reachable states of the system. Its goal is to expand the decision diagram that encodes the initial state node by node, in a bottom-up fashion, hence its name – *saturation* [CLS01; CMS06].

A crucial property required for the efficiency of saturation is called locality. The notion of locality means that when the system changes state in response to some event, only some of its components will actually transition to a new state, most of them will not be affected. If events are local, the exploration of the state space can be decomposed into smaller explorations on submodels with only a subset of components, considering only those events that are local on them. Saturation follows this strategy by considering a series of submodels: when *saturating* a decision diagram node, it considers only those variables whose values are yet to be considered in that subdiagram (and note those whose values led to the node) and explores the states reachable with the events local on these variables.

Saturation proved to be one of the most efficient decision diagram-based symbolic model checking algorithms, especially for concurrent systems. It was initially designed for Petri net models [CLS01], where it exploited the so-called Kroenecker condition of transitions (which allows for a very simple representation), but later improvements removed this constraint [CMS06]. A distinct variant of the algorithm is *constrained saturation* [ZC09], which can keep exploration inside a predefined set of states without modifying the transitions of the model (this problem appears e. g. when searching *backwards* from a state in an already explored state space). The problem with extending the transitions by an additional check to see if they leave the constraint set is that it destroys locality, because that check will depend on all state variables even when the original transition did not. With constrained saturation, the algorithm handles the constraint set separately and in this special case it can still exploit locality of the original transitions.

Besides the problems solved so far, there are a number of open questions related to the saturation algorithm. Even though constrained saturation solves the issue for constraint sets, it is generally hard for saturation to handle global or interdependent transitions. Such an interdependency is introduced when transitions have priorities. In this case, transitions have to consider also whether any other transition with a higher priority could be executed, which most of the time makes the transition global in the sense that it must be aware of all state variables. Prioritized transitions are common in Generalized Stochastic Petri Nets [Chi+93], which is a popular modeling formalism for stochastic systems. In this setting, symbolic model checking not only has to provide efficient state space exploration, but also has to support deriving a proper representation for numerical solvers.

Another challenge arises when saturation is used for the model checking of linear temporal logic (LTL) properties. LTL describes (infinite) executions of a system and translates to Büchi automata, an extension of finite state automata that accepts infinite words. Model checking of LTL properties involves the synchronization of the system model and the automaton translated from the negated LTL property such that the automaton reads the labels of system states as letters. If the automaton accepts an execution of the system then we have a proof that the property can be violated (as we translate the negated property). Since the automaton usually reads most of the state variables, the synchronized transitions will again lose locality. Furthermore, unlike explicit state graph-based model checkers, symbolic model checkers are usually not able to look for accepted executions before the state space is completely explored, whereas stopping upon finding a counterexample would be very much desirable.

In general, saturation is weak for systems where events are not local enough. In such cases, it will degrade to BFS or DFS, which tend to produce larger intermediate decision diagrams leading to larger resource consumption and less efficient scaling. The research presented in this work focuses on extending saturation to handle these situations efficiently. At the end, it turns out that there is a common idea in all of these extensions that can be used to further enhance saturation on many classes of models and also generalize many different variations that were considered as different algorithms before.

1.1 Summary of Challenges

Challenge 1: Extending saturation to efficiently handle Petri nets with priorities. Generalized Stochastic Petri nets (GSPN) are a popular formalism to model stochastic systems. The efficient state space exploration of GSPNs is an important part of stochastic analysis, which can verify extrafunctional properties of a system such as performance or reliability. Can the efficiency of saturation be leveraged for prioritized Petri nets?

Challenge 2: Extending saturation to efficiently perform model checking of linear temporal logic (LTL) properties. In the model checking of LTL properties, the system behavior has to be synchronized with a Büchi automaton describing the property. To use saturation, we have to avoid the direct computation of synchronized transition relations in order to avoid losing locality. Can we modify saturation to compute the synchronous product on the fly while also exploiting locality?

Challenge 3: Finding accepting executions (that violate the property) on the fly during state space exploration with saturation. In LTL model checking, we have to look for accepted executions (i. e. counterexamples) in the synchronous product of the system model and the Büchi automaton belonging to the negated property. In a finite state space, an accepting run is always a lasso, i. e. a path leading to a cycle. Therefore, the problem can be reduced to finding reachable strongly connected components (SCC) in the state space. Explicit-state model checkers can do this on the fly, stopping immediately when an accepted execution is found. Is there a way to create an efficient on-the-fly SCC detection algorithm and incorporate it into saturation-based model checking?

Challenge 4: Generalizing the different variants of constrained saturation. Constrained saturation is an efficient take on recovering locality in special cases. Its core idea to recover locality appears in many variants solving specialized problems. Currently, these variations are considered to be separate algorithms, which hinders the opportunity to freely combine different aspects of model checking (e. g. modeling and specification formalisms). Is there a way to generalize this idea as an abstract algorithm from which constrained saturation and other variants can be instantiated?

Challenge 5: Adapting saturation to better handle models where events have global effects. Some models have mainly synchronous behavior where every transition has to be aware of most of the components or variables. In this case, saturation will not be able to exploit locality and will degrade to less efficient exploration strategies. Is it possible to adapt the ideas of saturation even in these cases? Will the resulting improvements be beneficial on concurrent, asynchronous models where transitions are local and saturation is already efficient?

The research presented in this dissertation addresses these challenges and in particular aims to remove some of the limitations of saturation to provide even more efficient algorithms. Doing so expands the class of problems where model checking is applicable in practice and facilitates the spreading of automatic formal verification tools. To this end, the dissertation presents new scientific results in the form of three new saturation-based algorithms for model checking: saturation extended for prioritized models (Thesis 1), a complete algorithm family for the incremental, on-the-fly model checking of LTL properties with saturation (Thesis 2), and finally a generalization and enhancement of the saturation algorithm itself that incorporates the discovered ideas into the original algorithm to further improve its efficiency (Thesis 3). Table 1.1 presents how the results relate to the described challenges.

Table 1.1: Relations between theses and challenges.

		Challenge				
		1	2	3	4	5
Thesis 1	*Chapter 3 of the dissertation*	•			•	
Thesis 2	*Chapters 4–5 of the dissertation*		•	•	•	
Thesis 3	*Chapter 6 of the dissertation*				•	•

1.2 Contributions and Structure of the Dissertation

The contributions presented in this dissertation are organized into three theses based on the targeted field and are presented in five chapters loosely following the addressed challenges from Section 1.1.

- *Thesis 1* (Chapter 3) focuses on the efficient state space exploration of stochastic systems modeled as Generalized Stochastic Petri Nets, which is the first step of stochastic analysis. GSPNs have prioritized transitions that pose a challenge to the saturation algorithm because transitions cannot be evaluated locally. The efficient state space exploration of GSPNs can be a bottleneck of stochastic analysis, so advancements in this field will improve the whole process, leading to scalable stochastic analysis that is necessary to prove extra-functional requirements in safety-critical systems. The thesis will introduce Edge-valued Interval Decision Diagrams (EVIDD) as a compact encoding of priority-related aspects of the transition relation. It will also present a modified saturation algorithm that handles the priority-related part separately, achieving better scalability than previous approaches.

  This thesis responds to Challenge 1 discussed in Section 1.1.

- *Thesis 2* (Chapters 4–5) contains contributions for the model checking of linear temporal logic properties. Chapter 4 will present another modification of the saturation algorithm to handle the transition relation of the Büchi automaton (describing the property) separately. The solution builds on ideas of the constrained saturation algorithm to preserve the locality of transitions and improve the performance of the algorithm. Chapter 5 discusses the other aspect of LTL model checking related to finding counterexamples. I propose a complex approach built around a fixed point computation algorithm for computing strongly connected components in the state space, which is integrated with the saturation algorithm to be an incremental, on-the-fly symbolic model checking algorithm for LTL properties. The solution applies two heuristics that serve as cheap filters to reduce unnecessary computation (in situations where the absence of an SCC can be easily proved).

  This thesis responds to Challenges 2 and 3 discussed in Section 1.1.

- *Thesis 3* (Chapter 6) is about the generalization of the core ideas in Theses 1 and 2 as well as the constrained saturation algorithm. Work on the previous theses revealed common problems that appear in many different contexts: *1)* different next-state representations (including but not restricted to different types of decision diagrams) usually come with a specialized variant of the saturation algorithm and are not compatible with each other, as well as *2)* losing locality is generally a concern in every variant, whereas usually there is a workaround that retains some of the locality at least. I identified and introduced the notion of conditional locality that is a weaker requirement towards transition relations than locality, but is enough for the main idea of saturation to work. Chapter 6 introduces a generalized version of saturation based on conditional locality that outperforms the original saturation algorithm by orders of magnitudes in some cases, while it has no considerable overhead in any other case.

  This thesis responds to Challenges 4 and 5 discussed in Section 1.1.

1.3 Generalizability and Potential Impact

The proposed algorithms will be illustrated and evaluated on Petri net examples (see Section 2.1.2). However, all of them are applicable on a much wider range of models. Section 2.3.1 describes the precise requirements toward any modeling formalism to be compatible with the proposed approaches, and it allows virtually any finite-state model to be analyzed. Even the results of Thesis 1 – which were specifically tailored to GSPNs – is applicable and relevant on any (partitioned) transition system where high-level behaviors have priorities. This is often the case in state machines describing software or protocols, where priorities are a convenient way of avoiding nondeterministic behavior.

Model checking in general have a large potential technological and sociological impact. It fits in the general trend of using machines, and in particular software, to automate tedious and error-prone tasks (such as most of the advanced tools we use in the 21st century), as well as safeguarding and controlling processes where human error could lead to catastrophic results (such as power plants, driver assistance and most safety systems). Model checking and related techniques can become an invaluable assistant to software and system developers, automating quality-related tasks and augmenting testing, providing feedback and insights about the system-under-development and even suggesting solutions to fix problems or implement functionality (with automated model repair and synthesis techniques). Since the main limitation of these methods is scalability, any improvements in this regard will lead us closer to this future. Asynchronous concurrent systems, for which saturation is particularly efficient, are among the hardest both in getting them right and in analyzing them, therefore the proposed solutions may have an even higher impact.

Some of algorithmical innovations presented in the theses have potential applications outside of model checking. Computing the transitive closure of relations have applications in constraint programming [c15] and can be used anywhere where underlying graph structures are defined implicitly. Strongly connected components also play a key role in many domains where graphs are used, and the algorithms proposed in Thesis 2 may be applied to incrementally compute them during the traversal of the graph. The introduced ideas may also benefit related fields such as model synthesis, game theory and test generation.

A summary of applications and potential use cases of model checking and the proposed algorithms in particular is presented in Section 7.4.

CHAPTER 2

Background

The dissertation builds on well-established results from the fields of mathematical logic, graph theory, automata theory, as well as previously designed algorithms that are presented in this chapter as background knowledge or related work. Section 2.1 briefly introduces the basics of model checking, followed by a discussion of automata-theoretic (Section 2.2) and symbolic (Section 2.3) model checking. Finally, the saturation algorithm for state space generation is presented in depth as the direct basis of the algorithms proposed in the theses.

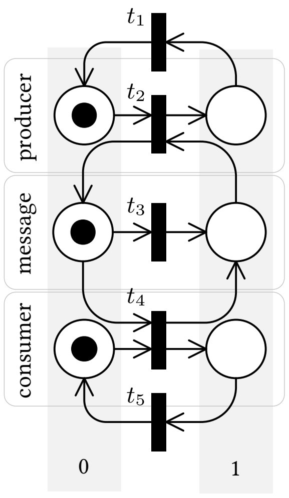
2.1 Introduction to Model Checking

Model checking is a formal verification method to verify properties of finite state systems, i. e. to decide whether a given formal model M satisfies a given requirement φ or not. The name comes from formal logic, where a logical formula may have zero or more models, which define the interpretation of the symbols used in the formula and the base set such that the formula is true. In this sense, the question is whether the formal model is indeed a model of the formal requirement: $M \vDash \varphi$?

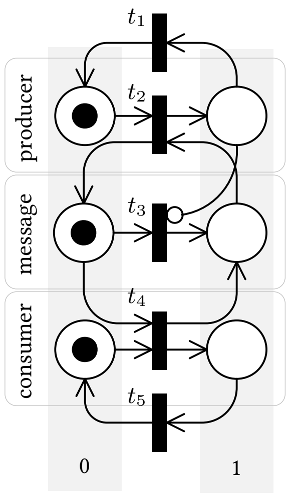
The formal model and the requirement can be given with many different formalisms. In the cases discussed in this dissertation, the formal model is given by a Kripke structure (Section 2.1.1) or by a high-level model that can be transformed into a Kripke structure, such as Petri nets (Section 2.1.2). Most of the algorithms require high-level models to be translated into a Kripke structure M, a process called state space generation (Section 2.1.3). Model checking itself is a collective name for algorithms evaluating $M \vDash \varphi$ (Section 2.1.4), where the requirement can be a simple safety property formalized as a state invariant in propositional logic (Section 2.1.4.1) or a temporal logic expression, i. e. a formula that expresses a temporal evolution of system states with a logical statement. One of the most used temporal logic formalisms is LTL, a specification language using linear (non-branching) logical time (Section 2.1.4.2).

2.1.1 Kripke Structures

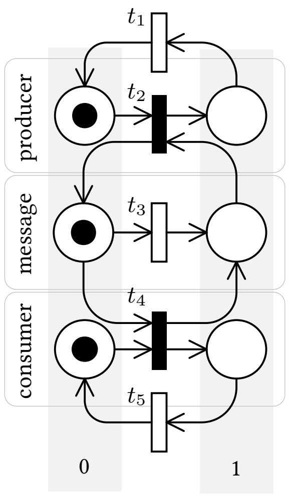
Kripke structures [Kri63] are directed graphs with labeled nodes. Nodes represent different states of the modeled system, while arcs denote state transitions. Each state is labeled with properties that hold in that state, called atomic propositions (atomic in the sense that they apply to one state in isolation). This way, paths in the graph represent possible behaviors of the system. Labels along the paths give the opportunity to reason about sequences of states through their properties given as Boolean propositions.



(a) Simple version.



(b) With inhibitor arc.



(c) With timed transitions.

Figure 2.1: Petri net models describing 3 variations of a producer-consumer system with a buffer for a single message. We treat each component as a Boolean variable which is 0 when the left place is marked and 1 when the right one.

Definition 1 (Kripke structure) Given a set of atomic propositions $AP = \{p, q, \ldots\}$, a (finite) Kripke structure is a 4-tuple $M = (\mathcal{S}, \mathcal{I}, \mathcal{N}, L)$, where:

- $\mathcal{S} = \{s_1, \ldots, s_n\}$ is the (finite) set of states;
- $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation consisting of state pairs (s_i, s_j);
- $L : \mathcal{S} \to 2^{AP}$ is the labeling function that maps a set of atomic propositions to each state. ▪

In the setting of LTL model checking, it is usually required that every state has at least one successor, i. e. there are no *deadlocks*. This requirement is captured by defining the transition relation as left-total, i. e. for all $s_i \in \mathcal{S}$, there exists $s_j \in \mathcal{S}$ such that $(s_i, s_j) \in \mathcal{N}$ (also denoted as $s_j \in \mathcal{N}(s_i)$). In that case, a *path* in M can be defined as an infinite sequence $\rho \in \mathcal{S}^\omega$ with $\rho(0) \in \mathcal{I}$ and $(\rho(i), \rho(i+1)) \in \mathcal{N}$ for every $i \geq 0$. The set of paths of a Kripke structure is denoted by $paths(M)$. The infinite sequence of sets of atomic propositions assigned to the states in ρ by L is called a *word* on the path and is denoted by $L(\rho) \in (2^{AP})^\omega$. The language described by a Kripke structure M (i. e. the set of all possible words on every path of M) is denoted by $\mathcal{L}(M)$ and is an ω-regular language (see Section 2.2.1). For systems that do have deadlocks, it is common to add self-loops to dead-end states, called *stuttering*, and restrict the LTL language to operators that are *stutter invariant* [Ben+14] (see Section 2.1.4.2).

2.1.2 Petri Nets

Petri nets are a widely used formalism to model concurrent, asynchronous systems [Mur89]. The formal definition of a Petri net (including inhibitor arcs) is as follows (see Figure 2.1 for an illustration of the notations).

Definition 2 (Petri net) A *Petri net* is a tuple $\mathrm{PN} = (P, T, W, M_0)$ where:

- P is the set of *places* (defining state variables);

- T is the set of *transitions* (defining behavior) such that $P \cap T = \emptyset$;
- $W = (W^- \sqcup W^+ \sqcup W^\circ)$ is a defined by three types of *arcs* (with their *weight function*), where $W^- : P \times T \to \mathbb{N}$, $W^\circ : P \times T \to \mathbb{N} \cup \{\inf\}$ and $W^+ : T \times P \to \mathbb{N}$ are the set of input arcs, inhibitor arcs and output arcs, respectively ($\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers);
- $M_0 : P \to \mathbb{N}$ is the *initial marking*, i. e. the number of *tokens* on each place. ▪

The three types of weight functions describe the structure of the Petri net: there is an input or output arc between a place p and a transition t iff $W^-(p,t) > 0$ or $W^+(t,p) > 0$, respectively, and there is an inhibitor arc iff $W^\circ(p,t) < \infty$.

The state of a Petri net is defined by the current marking $M : P \to \mathbb{N}$. The dynamic behavior of a Petri net is described as follows. A transition t is *enabled* iff $\forall p \in P : M(p) \in \big[W^-(p,t), W^\circ(p,t)\big)$. Any enabled transition t may fire non-deterministically, creating the new marking M' of the Petri as follows: $\forall p \in P : M'(p) = M(p) - W^-(p,t) + W^+(t,p)$. We denote the firing of transition t in marking M resulting in M' with $M \xrightarrow{t} M'$. A marking M_i is *reachable* from the initial marking if there exists a sequence of markings such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_i} M_i$. The set of reachable markings is denoted by $\mathcal{S}_r$. This work assumes $\mathcal{S}_r$ to be finite.

Petri Nets with Priorities Prioritized Petri nets extend the base formalism with a notion of transition priorities [Mur89], such that a prioritized Petri net is a tuple $\mathrm{PN} = (P, T, W, M_0, \pi)$ where $\pi : T \to \mathbb{N}$ assigns priorities to transitions. Priorities can be used to resolve conflicts between transitions: where unprioritized transitions would be chosen nondeterministically, different priorities can define a preference and lead to deterministic behavior. In this work, we follow the convention that priorities are non-negative and transitions with a higher priority have precedence over transitions with lower priority.

The dynamic behavior of prioritized Petri nets is extended with the notion of *fireable* transitions: a transition t is fireable iff it is enabled and there is no other enabled transition t' such that $\pi(t) < \pi(t')$. Only fireable transitions may fire, and the selection from fireable transitions is still nondeterministic (i. e. fireable transitions with the same priority will still fire nondeterministically).

Generalized Stochastic Petri Nets Stochastic Petri nets (SPN) extend Petri nets with timed behaviors, where transitions are equipped with exponentially distributed firing delay random variables [Ajm88]. Timed semantics of SPNs are defined by continuous-time Markov chains. Generalized Stochastic Petri nets (GSPN) further extend modeling capabilities to support both timed and instantaneous behaviors [Chi+93]. In GSPNs, transitions with zero priority (called *timed*) have exponentially distributed firing delays, while transitions with $\pi(t) \geq 1$ are *immediate*. Figure 2.1c presents an example for a GSPN, where filled rectangles are immediate (untimed) transitions and empty rectangles are timed transitions.

A prioritized Petri net marking M where no transition t with $\pi(t) \geq 1$ is enabled is called *tangible*, while markings with an enabled transition t with $\pi(t) \geq 1$ are called *vanishing*. We write $M \in \mathcal{T}$ if $M \in \mathcal{S}_r$ is a reachable tangible marking and $M \in \mathcal{V}$ if M is a reachable vanishing marking. In tangible markings, the timed semantics of Stochastic Petri nets apply to GSPNs. In contrast, immediate transitions are fired in vanishing markings while no time elapses. Conflicts between immediate transitions may yield nondeterministic behaviors. To ensure that probability distribution of GSPN markings evolve deterministically in time, conflicts must be resolved by assigning *probability weights* and priorities [Chi+93]. Conflict resolution may yield prioritized Petri nets with many priority levels [TFP03].

Running Example Figure 2.1 presents three Petri net models of a producer-consumer system with a single message passed back and forth between them. The left-most model (2.1a) is a simple version where both the producer, the consumer and the message have two *places* (the circles), representing their two possible states: *ready* and *busy* in case of the producer and consumer and *unprocessed* and *processed* in case of the message. When a place is marked by a *token* (black filled circle), the component is assumed to be in the corresponding state. States may be changed by *transitions* (rectangles). *Arcs* define nonzero weights in the weigh functions (weight of 1 unless written on the arc).

That said, the example model has 5 possible behaviors: t_1 and t_5 will return a *busy* producer or consumer to the *ready* state, t_2 will cause a *ready* producer to remove the *processed* message and create a new, *unprocessed* one, while the producer will become *busy* with producing the next message, t_4 similarly causes the consumer to consume and process an *unprocessed* message, and t_3 represents a timeout when the *unprocessed* message simple becomes *processed.* With these transitions, we can see that exactly one place of every component will be marked in any state of the system, so we may treat each component as a Boolean variable: it is 0 when the left place is marked and 1 when the right one.

The model in the middle (2.1b) extends the system with an *inhibitor arc* (an arc with a circle instead of an arrow). Inhibitor arcs will *disable* a transition if the connected place is marked (by default by one token). Therefore, this variant says that a timeout may only occur if the producer is *ready*, e. g. because the producer's process keeps track of timeouts and will only check when it is *ready.*

The third model (2.1c) uses the Generalized Stochastic Petri Net (GSPN) formalism. In this formalism, we can distinguish two types of transitions: *immediate* transitions are just like before and will fire as soon as they are enabled (possibly in a nondeterministic order), while *timed* transitions (empty rectangles) have a *firing rate* that affects how soon they will fire after becoming enabled. An important consequence of this distinction is that immediate transitions *must* fire before any enabled timed transition, adding *priorities* to the transitions.

With this in mind, the third model adds the information that the producer and consumer will actually spend time on creating and processing messages (*busy* state), and the timeout will happen after a specific amount of time, while the emitting (t_2) and consuming (t_4) of new messages will happen immediately when possible. A consequence of the implied priorities is that e. g. a timeout may not occur if the consumer is ready to consume a message.

2.1.3 State Space Generation

While model checking algorithms work on low-level formalism and modeling is usually done in a high-level modeling language, practical applications must address the translation from high-level to low level. In this work, we use the term *state space generation* to refer to the process of computing a Kripke structure that describes the *reachable* state space of a high-level model – in this case, Petri nets. Therefore, if not implied otherwise, the term *state space* will refer to a Kripke structure in which every state is reachable from some of the initial states, while *potential state space* will refer to an overapproximation that is a superset of the reachable state space and by convention contain all states that can be set as initial state in the high-level formalism. When speaking about state graphs, we mean the pair $(\mathcal{S}, \mathcal{N})$ (states and transitions of a Kripke-structure) interpreted as a graph. As an example, we define the state space generation problem for Petri nets.

Definition 3 (State space of a Petri net as a Kripke structure) Given a set of atomic propositions AP reasoning about markings of a (prioritized) Petri net PN, the state space of PN is a (finite) Kripke structure with:

- $\mathcal{S} = \mathcal{S}_r$ is the set of reachable markings of PN (as defined in Section 2.1.2);
- $\mathcal{I} = \{M_0\}$ is the initial marking of PN;
- $\mathcal{N} = \{(M, M') \mid M \in \mathcal{S} \wedge M' \in \mathcal{S} \wedge \exists t \in T : M \xrightarrow{t} M'\}$ is the transition relation between reachable markings defined by transitions of PN;
- $L(M) = \{p \mid M \vDash p\} \subseteq AP$, i. e. each marking is labeled with the atomic propositions that are true in that marking. ▪

Note that *transition* is used with two different meanings: whenever ambiguous, we will use *low-level transition* to refer to an element of a transition relation of a Kripke structure and *high-level transition* to refer to transitions of a Petri net. The relationship is one-to-many, such that a single high-level transition can be regarded as an *event* that can happen in the high-level model, resulting in one of potentially many different low-level (state-)transitions depending on the current state.

A non-trivial part of state space generation is the computation of reachable states (markings), which can be formally expressed as the problem of computing $\mathcal{N}*(\mathcal{I})$, where $\mathcal{N}*$ is the reflexive-transitive closure of the transition relation. The term state space generation is often applied to this problem only, with algorithms focusing on the efficient traversal and representation of the state space. Graph traversal algorithms such as breadth-first search and depth-first search can be used to generate the state space, but the large size of the explicit state graph (known as the *state space explosion problem*) often prevents the practical application of these simpler algorithms. Symbolic or "implicit" model checking addresses this problem by the compact encoding of sets and relations and using set operations instead of graph algorithms (see Section 2.3). Other approaches such as partial order reduction [Pel98] and counterexample-guided abstraction refinement [Cla+00] try to avoid the full computation of the state space by omitting information that is insignificant with regard to the property to be checked.

Especially in relation to abstractions, we have to mention that deciding whether a label is on a state or not can be a hard problem. In general, the label has to be derivable from the information known about the state in the high-level model, which requires automated theorem proving – an undecidable problem. Nevertheless, this work will assume that checking whether a state s is labeled with an atomic proposition $p \in AP$ (i. e. $p \in L(s)$) is decidable in $\mathcal{O}(1)$. This is a good abstraction that lets us focus on the additional complexity of the model checking algorithms and can be close to reality if the logic used in atomic propositions and the information known about a state are simple (e. g. quantifier-free first order logic with the theory of integers and the exact marking of a Petri net).

2.1.4 Model Checking

Given a requirement φ usually expressed in some kind of logic, the main goal of model checking is to evaluate $M \vDash \varphi$ where M is the result of state space exploration. If the evaluation happen simultaneously with state space exploration, the algorithm is said to do *on-the-fly* model checking. This has an advantage that the exploration – which is usually the most expensive part – can be stopped as soon as there is a result. The result itself is a decision about $M \vDash \varphi$, and algorithms usually provide a *witness* for at least one of the outcomes ($\vDash$ or $\nvDash$) that proves the answer and can be checked easily. In general, if the property is universal, the witness will be a counterexample for the negative results (proving that the model violates the property), while for existential properties the witness is an example proving the positive result (the model satisfies the property). The following sections present state invariants and linear temporal logic, defining their semantics in terms of Kripke structures that provides the basis for model checking algorithms.

2.1.4.1 State Invariants

State invariants are one of the simplest universal properties that can be verified with model checking [CGP99]. A state invariant is a logic expression characterizing a set of states on which it evaluates to true. We require that state invariant must be true in every reachable state of the system, or equivalently, no state is reachable where the state invariant does not hold. Formally, given a set of atomic propositions AP, a state invariant $\varphi \in AP$ and a Kripke sturcture M, $M \models \varphi$ if $\forall s \in \mathcal{S}$, the state satisfies φ: $s \models \varphi$ i. e. $\varphi \in L(s)$. A common example for a state invariant in programming is an *assertion*, which can be written as an implication: if the program counter points to an assertion instruction, the expression in the assertion must be true. In safety-critical systems, a common requirement that is a state invariant is to avoid dangerous states, or equivalently, sustain safe behavior.

Verifying state invariants is essentially equivalent to state space generation, with special emphasis on evaluating atomic propositions (which are usually defined implicitly and computed on demand). It is also equivalent to verifying reachability, which is an existential property, and can be expressed as a state invariant indirectly by assuming that the specified states are not reachable and negating the result. Assuming that deciding whether a state is labeled with an atomic proposition can be done in $\mathcal{O}(1)$, the complexity of checking reachability is $\mathcal{O}(|\mathcal{S}|)$, which is usually dominated by the complexity of state space generation.

2.1.4.2 Linear Temporal Logic

Linear temporal logic (LTL) [Pnu77] is a temporal logic that describes the temporal evolution of state labels during an execution. Its time model is linear in the sense that it considers a single realized future behavior of a system, i. e. a single path in a Kripke structure (as opposed to computational-tree logic (CTL), which has a branching time model and reasons about what is possible from a state instead of what should happen during the execution). The operators in LTL are the following: X (in the *neXt state*), F (in the Future, or more precisely, *eventually*), G (*Globally*), U (*Until*) and R (*Release*). LTL without the X operator is *stutter invariant* (see Section 2.1.1).

Definition 4 (Syntax of LTL) The formal syntax of LTL is given by the following grammar in Backus–Naur Form (BNF), where $p \in AP$ is an atomic proposition:

$$\phi ::= \top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \mathsf{X}\phi \mid \mathsf{F}\phi \mid \mathsf{G}\phi \mid [\phi \ \mathsf{U} \ \phi] \mid [\phi \ \mathsf{R} \ \phi]$$

A well-formed LTL formula is generated by ϕ. ▪

An LTL formula φ is said to be *valid* with regard to a Kripke structure M, if it holds for all paths of M (in this case it is a universal property regarding the set of paths, which can be refuted by a counterexample). It is *satisfiable* if it holds for some path in M (as an existential property that can be proven with an example). As a specification language, usually validity is desired.

Definition 5 (Semantics of LTL) The formal semantics of LTL is defined with respect to a Kripke structure M with a left-total transition relation (i. e. every path ρ of M is infinite). Let ρ be a path of M, and let $\rho[k]$ be a suffix of ρ starting from element k. The relation $\rho \models \phi$ is defined inductively:

1. $\rho \models \top$;
2. $\rho \models p$ iff $p \in L(\rho(0))$;
3. $\rho \models \neg\phi$ iff $\rho \not\models \phi$;

4. $\rho \vDash \phi_1 \wedge \phi_2$ iff $\rho \vDash \phi_1$ and $\rho \vDash \phi_2$;
5. $\rho \vDash \mathsf{X}\phi$ iff $\rho[1] \vDash \phi$;
6. $\rho \vDash [\phi_1 \mathsf{U} \phi_2]$ iff for some $k \geq 0$, $\rho[k] \vDash \phi_2$ and for all $0 \leq i < k$, $\rho[i] \vDash \phi_1$.

Other Boolean operators and symbols are defined as usual based on $\top$, $\neg$ and $\wedge$. The remaining temporal operators can be defined with the following equivalences:

1. $[\psi \mathsf{R} \varphi] \equiv \neg[\neg\psi \mathsf{U} \neg\varphi]$
2. $\mathsf{F} \varphi \equiv [\top \mathsf{U} \varphi]$
3. $\mathsf{G} \varphi \equiv [\bot \mathsf{R} \varphi]$ ▪

As the definition of the semantics suggests, LTL can distinguish between words on paths of a Kripke structure, which are ω-regular words over 2^{AP} as discussed in Section 2.1.1. This way, it is possible to speak about such words satisfying an LTL formula φ, denoted by $w \vDash \varphi$, and the language of the formula $\mathcal{L}(\varphi) = \{w \mid w \vDash \varphi\}$, which is again an ω-regular language. In fact, the language of any LTL formula is an ω-star-free language, which is a strict subset of ω-regular languages [Coh91]. The model checking of LTL properties is much more complex than verifying state invariants, so it is discussed separately in the next section.

2.2 Automata-Theoretic Model Checking

Sections 2.1.1 and 2.1.4.2 already discussed words and language in relation to paths of a Kripke structure, and mode generally to an execution of a system. Automata-theoretic (LTL) model checking approaches are based on this language-centric formalization of systems and properties and usually build on the theory of Büchi automata and especially their synchronous product, both discussed in this section.

An *automaton* is an abstract machine that models some kind of computation over a sequence of input symbols. In formal language theory, automata are mainly used as a finite representation of infinite languages. In that context, they are often classified by the class of languages they are able to recognize.

The simplest class of automata is finite automata (also known as finite state machines). A finite automaton operates with a finite and constant amount of memory (independent of the length of the input). A finite automaton can operate on finite or infinite inputs. In formal language theory, inputs are called words, while elements of the input are called letters, the set of all possible letters constituting the alphabet.

A simple type of finite automata reading infinite words is the so-called Büchi automaton, which typically plays a key role in LTL model checking. This section overviews the definition of Büchi automata, the ways to represent other formalisms as an equivalent Büchi automaton and the synchronous product of such automata, then concludes with the summary of automata-theoretic LTL model checking.

2.2.1 Büchi Automata

Büchi automaton [Büc62] is one of the simplest finite automata operating on infinite words.

Definition 6 (Büchi automaton) A Büchi automaton is a 5-tuple $A = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where:

- $\Sigma = \{\alpha_1, \dots, \alpha_n\}$ is the finite alphabet;
- $\mathcal{Q} = \{q_1, \dots, q_m\}$ is the finite set of states;
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states;

- $\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation consisting of state–input–state triples (q, α, q');
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states. ▪

A *run* of a Büchi automaton *reading* an infinite word $w \in \Sigma^\omega$ is a sequence of automaton states $\rho_w \in \mathcal{Q}^\omega$, where $\rho_w(0) \in \mathcal{I}$ and $(\rho_w(i), w(i), \rho_w(i+1)) \in \Delta$ for all i, i. e. the first state is an initial state and state changes are permitted by the transition relation. Let $\mathit{inf}(\rho_w)$ denote the set of states that appear infinitely often in ρ_w. The run ρ_w is called *accepting* iff it has at least one accepting state appearing infinitely often, i. e. $\mathit{inf}(\rho_w) \cap \mathcal{F} \neq \emptyset$. A Büchi automaton A accepts a word w iff it has an accepting run reading w. The set of all infinite words accepted by a Büchi automaton A is denoted by $\mathcal{L}(A) \subseteq \Sigma^\omega$ and is called the *language* of A. The class of languages that can be characterized by Büchi automata is called ω-regular languages.

An important question about a Büchi automaton A is whether it accepts any words, i. e. if $\mathcal{L}(A) = \emptyset$. Because of the finite set of states and infinitely long words, accepted words will always correspond to a lasso-shaped run of the Büchi automaton: the cycle part of the lasso will contain an accepting state which is therefore visited infinitely many times and there will be a path from an initial state to the cycle. Therefore, checking language emptiness can be reduced to looking for reachable cycles, or more generally, *strongly connected components* (SCC) in the state graph of the Büchi automaton that include at least one accepting state, often called *fair cycles* or *fair SCCs*. A suitable algorithm for this problem that is also used in Thesis 2 in Chapter 5 is Tarjan's algorithm [Tar72].

2.2.2 Kripke Structures and Büchi Automata

Although Kripke structures and Büchi automata are very similar in structure, Kripke structures are less expressive in terms of language, i. e. there are ω-regular languages that Kripke structures cannot produce. The following proposition presents a way to construct a Büchi automaton that accepts exactly the language of a Kripke structure.

Proposition 1 (Büchi automaton of Kripke structure) Given a Kripke structure $M = \langle \mathcal{S}, \mathcal{I}, \mathcal{N}, L \rangle$ with the set of atomic propositions AP, an equivalent Büchi automaton that accepts exactly the language produced by M is $A_M = \langle \Sigma, \mathcal{Q}, \mathcal{I}, \Delta, \mathcal{F} \rangle$, where:

- $\Sigma = 2^{AP}$, i. e. letter are sets of atomic propositions;
- $\mathcal{Q} = \mathcal{S} \cup \{\mathit{init}\}$, i. e. the states of the Kripke structure together with a special initial state;
- $\mathcal{I} = \{\mathit{init}\}$, i. e. the special initial state;
- $\Delta = \{(s, \alpha, s') \mid (s, s') \in \mathcal{N} \wedge \alpha = L(s')\} \cup \{(\mathit{init}, \alpha, s) \mid s \in \mathcal{I} \wedge \alpha = L(s)\}$, i. e. the automaton reads the labels of target states and additional transitions go from the special initial state to initial states of the Kripke structure[1];
- $\mathcal{F} = \mathcal{Q}$, i. e. every state is accepting. ▪

Defining accepting states as the entire set of states is indeed necessary, because every path in a Kripke structure produces a word of its language, no matter what states it passes. A general Büchi automaton, in contrast, can use accepting states to restrict its language to a subset of all the paths. Based on that, an example ω-regular language that cannot be produced by a Kripke structure is $a^* b^\omega$ (infinitely many b's after finitely many a's). Any Büchi automaton accepting this language must have a loop that reads the letter a arbitrary many times and another loop that reads b infinitely many times. Only the second loop can contain an accepting state to force runs out of the first loop, which cannot

[1]Technically, the special initial state is necessary to make the automaton read the labels of the initial states of the Kripke structure. The first step of the automaton can be regarded as the initialization of the corresponding Kripke structure.

be modeled in Kripke structures: if something can happen arbitrary many times in a Kripke structure, it can also happen infinitely many times.

2.2.3 LTL to Büchi Automata

Since linear temporal logic formulas characterize a strict subset of ω-regular languages (see Section 2.1.4.2), there is an equivalent Büchi automaton for every LTL formula φ that accepts the same language that satisfies φ. In the history of model checking, many approaches have been developed to perform this conversion.

In general, the resulting automaton can be exponential in the size of the LTL formula, but efficient algorithms exist that are applicable in practice, where formulas are usually small. The algorithm described in [Ger+95] was one of the first solutions, and there are some more advanced variants as well, such as [GO01].

Since atomic propositions in φ refer to labels of a Kripke structure, the alphabet of the equivalent automaton will be $\Sigma = 2^{AP}$. This is the same alphabet as that of the automaton describing the Kripke structure itself as defined in Proposition 1.

2.2.4 Synchronous Product of Büchi Automata

The synchronous product of two Büchi automata A_1 and A_2 over the same alphabet Σ is another Büchi automaton $A_1 \cap A_2$ that accepts exactly those words that both A_1 and A_2 accept [CGP99]. The language of the product automaton is therefore $\mathcal{L}(A_1 \cap A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. The construction of such synchronous product automata is well-known and can be found in e. g. [CGP99].

However, the setting of LTL model checking is special, as in the automaton of the Kripke structure all states are accepting ($\mathcal{F}_1 = \mathcal{Q}_1$, see Proposition 1). In such a case, the definition of the product automaton is simpler. Since any infinite run in A_1 will be accepting, the product inherits the accepting states of A_2.

Definition 7 (Synchronous product – special case) Given two Büchi automata $A_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{I}_1, \Delta_1, \mathcal{F}_1 \rangle$ and $A_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{I}_2, \Delta_2, \mathcal{F}_2 \rangle$ with $\mathcal{F}_1 = \mathcal{Q}_1$, their synchronous product is $A_1 \cap A_2 = \langle \Sigma, \mathcal{Q}_\cap, \mathcal{I}_\cap, \Delta_\cap, \mathcal{F}_\cap \rangle$, where:

- $\mathcal{Q}_\cap = \mathcal{Q}_1 \times \mathcal{Q}_2$, i. e. product states are pairs;
- $\mathcal{I}_\cap = \mathcal{I}_1 \times \mathcal{I}_2$, i. e. every combination of the initial states will be considered;
- $\Delta_\cap = \{((q,r),\alpha,(q',r')) \mid (q,\alpha,q') \in \Delta_1 \land (r,\alpha,r') \in \Delta_2\}$, i. e. both automata can process the input;[2]
- $\mathcal{F}_\cap = \mathcal{F}_1 \times \mathcal{F}_2 = \mathcal{Q}_1 \times \mathcal{F}_2$, i. e. accepting states of A_2 are inherited. ▪

Note that in this case, $\mathcal{Q}_\cap$ is a *potential* state space because state combinations are not necessary reachable from the initial states. The main challenge is the computation of $\Delta_\cap \subseteq (\mathcal{Q}_1 \times \mathcal{Q}_2) \times \Sigma \times (\mathcal{Q}_1 \times \mathcal{Q}_2)$ that is necessary if reachable states of the product are sought. The first part of Thesis 2 (Chapter 4) proposes an algorithm to compute reachable states efficiently even for large automata.

2.2.5 LTL Model Checking

Formally, the problem of LTL model checking is deciding if $\forall \rho \in paths(M) : \rho \vDash \varphi$, where M is a Kripke structure modeling a system and φ is an LTL formula describing a desired property of the

[2]In the context of LTL model checking, labels on the two synchronized transitions (q,α_1,q') and (r,α_2,r') do not have to be the same. We require only that $\alpha_1 \vDash \alpha_2$, which – according to Definition 1 – translates to $\alpha_2 \subseteq \alpha_1$, similarly to the model checking of state invariants.

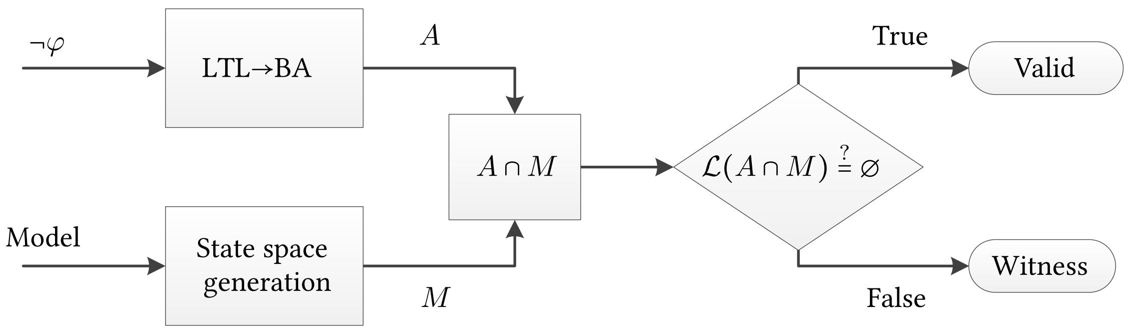


Figure 2.2: Steps of ω-regular model checking (with simplified notations).

system [EC80]. The essential idea of LTL model checking is to use Büchi automata to describe both the property and the system model (see Sections 2.2.2 and 2.2.3) and reduce the model checking problem to language emptiness [CGP99].

Given a Kripke structure M and an LTL formula φ using the same atomic propositions AP, let $\mathcal{L}(M)$ and $\mathcal{L}(\varphi)$ denote the languages produced by the Kripke structure and characterized by the formula. The language $\mathcal{L}(M)$ contains every observable behavior of M in terms of AP (provided behaviors), while $\mathcal{L}(\varphi)$ contains valid behaviors. The model checking problem can be rephrased to the following: is the set of provided behaviors fully within the set of valid behaviors? This is called the language inclusion problem and can be formalized as the decision of $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$.

An equivalent formalization is $\mathcal{L}(M) \cap \overline{\mathcal{L}(\varphi)} = \varnothing$, where $\overline{\mathcal{L}(\varphi)}$ is the complement of the language of φ. Although both the language inclusion problem and the complementation of Büchi automata are hard, in case of LTL model checking, the complementation can be avoided. The key observation is that the complement of the language of an LTL formula is the language of the negated formula: $\overline{\mathcal{L}(\varphi)} \equiv \mathcal{L}(\neg\varphi)$. This way, the model checking problem is reduced to language intersection and language emptiness, both of which can be efficiently computed on Büchi automata.[3]

This approach is called *automata-theoretic model checking* [Var96]. Given a high-level model and a linear temporal logic specification, the following steps have to be realized (see Figure 2.2):

1. Compute the Kripke structure M of the high-level model (Section 2.1.3).
2. Transform the Kripke structure into a Büchi automaton A_M (Section 2.2.2).
3. Transform the negated LTL formula into a Büchi automaton $A_{\neg\varphi}$ (Section 2.2.3).
4. Compute the synchronous product $A_M \cap A_{\neg\varphi}$ (Section 2.2.4).
5. Check language emptiness of the product: $\mathcal{L}(A_M \cap A_{\neg\varphi}) = \varnothing$ (Section 2.2.1).

If $\mathcal{L}(A_M \cap A_{\neg\varphi}) = \varnothing$ then the model meets the specification. Otherwise words of $\mathcal{L}(A_M \cap A_{\neg\varphi})$ are counterexamples, i. e. provided behaviors that violate the specification. The complexity of LTL model checking is $\mathcal{O}(|\mathcal{S}| \cdot 2^{|\varphi|})$, where $|\varphi|$ is the number of temporal operators in φ [Sch03].

Sometimes, it is possible to design the algorithms such that some steps may overlap or can be executed together. For example, many algorithms compute the product automaton on-the-fly during state space generation, using the high-level model as an implicit representation of the Kripke structure. Language emptiness may sometimes also be checked continuously during the computation of the product. If both optimizations are present in an algorithm, it is said to perform the LTL model checking on the fly (during state space generation) [Cou+91]. For an example, see [Ger+95] that is the base of the SPIN explicit model checker.

[3]This "trick" is possible because LTL cannot express every ω-regular language, therefore the problem of complementing the language of LTL formulas can be easier than for a Büchi automata.

2.3 Symbolic Model Checking

Symbolic model checking techniques have been devised to combat the state space explosion problem [McM92; Bur+92]. While traditional *explicit-state model checkers* employ graph algorithms and enumerate states and transitions one-by-one, *symbolic model checking* algorithms use a special encoding to efficiently represent the state space and try to process large numbers of similar states together. The special encoding can be regarded as a compression of the sets and relations, but unlike traditional data compression methods, the encoded data can be manipulated without returning to the explicit representation.

Symbolic model checking was first introduced for hardware model checking, where states of hardware models were encoded in binary variables [Cla+96]. Similarly, states of a Kripke structure can be encoded in at least $k = \lceil log(|\mathcal{S}|) \rceil$ Boolean variables. Sets of states can then be represented by Boolean *characteristic functions* $f_{\mathcal{S}} : \mathbb{B}^k \rightarrow \mathbb{B}$, returning true when a state is in the set, where $\mathbb{B} = \{\top, \bot\}$ is the set of Boolean values (true and false, respectively). The transition relation can also be represented by functions mapping from $2k$ binary variables to true or false, half of them encoding the source state, the other half encoding the target: $f_{\mathcal{N}} : \mathbb{B}^k \times \mathbb{B}^k \rightarrow \mathbb{B}$.

The functions are usually represented by Boolean formulas or binary decision diagrams (BDDs) [Bry86]. In case of Boolean formulas, the model checking problem is reduced to the Boolean satisfiability problem (SAT), while in case of decision diagrams, efficient algorithms are known to manipulate the sets directly in the encoded form [Bry86]. Based on these operations, model checking can be reduced to fixed point computations on sets of states, such as in the case of saturation algorithm presented in Section 2.4.

This concept can also be used with integers or other data types if the states encode information from a high-level model as state vectors. In that case, formulas will be in quantifier-free first order logic solved by SMT (satisfiability modulo theories) solvers, while decision diagrams will be extended to e. g. Multi-valued Decision Diagrams (MDDs) [MD98], which will be presented in Section 2.3.3.

2.3.1 Partitioned Transition Systems

To capture the structure of high-level models for symbolic encoding, algorithms like saturation prefer to use *partitioned transition systems* (PTS) instead of Kripke structures. In a PTS, state variables of the high-level model, high-level events (causing transitions) and their dependencies on state variables are preserved to partition the low-level next-state relation and localize the effect of transitions [CMS06]. In decision diagram-based model checking, such models usually come with a user-specified variable ordering.

Definition 8 (Variable ordering) A variable ordering over variables V ($|V| = K$) is a total ordering of elements of V that defines a sequence. The variable in position k of the sequence is denoted by x_k. We will say that x_1 is the *lowest* and x_K is the *highest* in the ordering. We will use the notations $V_{\leq k} = \{x_1, \dots, x_k\}$ and $V_{>k} = \{x_{k+1}, \dots, x_K\}$ for sets of variables constituting a prefix or suffix (respectively) of the sequence. ▪

With a specified variable ordering, the formal definition of a PTS is as follows.

Definition 9 (Partitioned Transition System) A partitioned transition system is a tuple $M = (V, D, \mathcal{I}, \mathcal{E}, \mathcal{N})$ where:

- $V = \{x_1, \dots, x_K\}$ is the finite set of K variables with an arbitrary but well-defined variable ordering;
- D is the domain function such that $D(x_k) \subseteq \mathbb{N}$ for all $x_k \in V$;
- $\mathcal{I} \subseteq \hat{S}$ is the set of initial states, where $\hat{S} = D(x_1) \times \dots \times D(x_K)$ is the potential state space (the shape of which is unaffected by the chosen variable ordering);
- $\mathcal{E}$ is the set of high-level events, specifying groups of individual transitions;
- $\mathcal{N} \subseteq \hat{S} \times \hat{S}$ is the next-state relation partitioned by $\mathcal{E}$ such that $\mathcal{N} = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e$. We often use $\mathcal{N}$ as a function returning the "next states": $\mathcal{N}(\mathbf{s}) = \{\mathbf{s}' | (\mathbf{s}, \mathbf{s}') \in \mathcal{N}\}$ and $\mathcal{N}(S) = \bigcup_{\mathbf{s} \in S} \mathcal{N}(\mathbf{s})$ for $S \subseteq \hat{S}$.

A (concrete) state of the system is a vector $\mathbf{s} \in \hat{S}$, where each variable x_k has a value from the corresponding domain: $\mathbf{s}[k] \in D(x_k)$. ▪

PTSs are halfway between high-level models and low-level models and are often used as abstractions of high-level models on which state space generation can be performed (note that the set of reachable states is not part of the definition of a PTS). To illustrate this, we will give the translation of Petri net to PTSs, as well as from PTSs to Kripke structures (including the computation of the set of reachable states).

Definition 10 (Petri net as PTS) Given a Petri net PN $= (P, T, W, M_0)$, the corresponding PTS representation is defined as $M = (V, D, \mathcal{I}, \mathcal{E}, \mathcal{N})$ such that:

- $V = P$, i. e. there is a variable for each place;
- $D : V \to \mathbb{N}$, i. e. each variable is a non-negative integer;
- $\mathcal{I} = M_0$, i. e. the initial state is the initial marking;
- $\mathcal{E} = T$, i. e. each transition in the Petri net is an event;
- $\mathcal{N} = \bigcup_{t \in \mathcal{E}} \mathcal{N}_t$ such that $\mathcal{N}_t = \{(M, M') \mid M \xrightarrow{t} M'\}$, i. e. the next-state relation is partitioned by transitions of the Petri net. ▪

Definition 11 (State space of a PTS as a Kripke structure) Given a set of atomic propositions AP reasoning about state vectors of a PTS $M = (V, D, \mathcal{I}, \mathcal{E}, \mathcal{N})$, the state space of M is a (finite) Kripke structure $M = (\mathcal{S}, \mathcal{I}, \mathcal{N}, L)$ with:

- $\mathcal{S} = \mathcal{N} * (\mathcal{I})$ is the set of reachable states from the set of initial states (as defined in Section 2.1.3);
- $L(\mathbf{s}) = \{p \mid \mathbf{s} \vDash p\}$, i. e. each state vector is labeled with the atomic propositions that are satisfied by that vector. ▪

As Definition 10 suggests, translating to a PTS is mostly syntactic, except sometimes the next state relation, depending on the representation. On the other hand, moving from PTS to Kripke structure (i. e. the state space) involves the computation of reachable states. Therefore, PTS is a suitable input formalism to symbolic model checking algorithms.

2.3.2 Abstractions

Symbolic model checking is closely related to abstraction-based techniques both in many SAT/SMT solver-based approaches (i. e. CEGAR [Cla+00]) and decision diagram-based approaches. In this section, we characterize the types of abstractions and similar techniques used throughout the dissertation.

Definition 12 (Abstraction of states) Given a set of concrete states S (with possible unreachable states) and a set of abstract states S', $f : S \to S'$ is an abstraction if it is a total surjective function, i. e. $\forall s \in S : f(s) \in S'$ and $\forall s' \in S'$ we have that $\exists s \in S : f(s) = s'$. We will use the notation $s \subseteq s'$ to say that s is a concrete state mapped to s', i. e. $f(s) = s'$.[4]

With this definition for abstraction, it is possible to define multiple variants for the abstraction of Kripke structures. Following [God14], we will build on *may* and *must abstractions*.

Definition 13 (May and Must abstraction of a Kripke structure) Assume a set of atomic propositions AP, a "concrete" Kripke structure $M' = (\mathcal{S}', \mathcal{I}', \mathcal{N}', L)$, an "asbtract" Kripke structure $M = (\mathcal{S}, \mathcal{I}, \mathcal{N}, L')$ and a total surjective function $f : \mathcal{S} \to \mathcal{S}'$ such that:

- $\mathcal{I}' = f(\mathcal{I})$, i. e. abstract initial states are the images of concrete initial states;
- $\forall s' \in \mathcal{S}' : L'(s') = \{p \mid \exists s \in \mathcal{S} : f(s) = s', p \in L(s)\}$, i. e. an abstract state gets all the labels of any concrete state mapped onto it;
- $\mathcal{N}' \subseteq \mathcal{S}' \times \mathcal{S}'$, i. e. abstract transitions are between abstract states.

M' is a *may abstraction* of M if:

- $(s'_1, s'_2) \in \mathcal{N}' \Rightarrow \exists s_1 \subseteq s'_1 : \big(\exists s_2 \subseteq s'_2 : (s_1, s_2) \in \mathcal{N}\big)$, i. e. if there is a transition from abstract state s'_1 to s'_2 then *there exists* a concrete state s_1 mapped to s'_1 from which there is a transition to any concrete state s_2 mapped to s'_1.

M' is a *must abstraction* of M if:

- $(s'_1, s'_2) \in \mathcal{N}' \Rightarrow \forall s_1 \subseteq s'_1 : \big(\exists s_2 \subseteq s'_2 : (s_1, s_2) \in \mathcal{N}\big)$, i. e. if there is a transition from abstract state s'_1 to s'_2 then *for every* concrete state s_1 mapped to s'_1 there is a transition to any concrete state s_2 mapped to s'_1. ▪

A may abstraction M' can simulate the concrete Kripke structure M in the sense that for any path ρ of M there is a path ρ' in M' such that $\forall i \geq 0 : f\big(\rho(i)\big) = \rho'(i)$, i. e. abstract states of the abstract path are abstractions of the concrete states of a concrete path. With the languages of the Kripke structures, this can be written as $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ – whatever happens in the may abstraction *may* (or may not) happen in the concrete model, but nothing else.[5] On the other hand, a *must abstraction* M'' can be simulated by M, i. e. $\mathcal{L}(M'') \subseteq \mathcal{L}(M)$ – what happens in the must abstraction *must* have an equivalent in the concrete model (and more things can happen, too).

A special case of may abstraction on PTSs is *projection*, which is implied by the surjective function of projecting state vectors to a subset of variables.

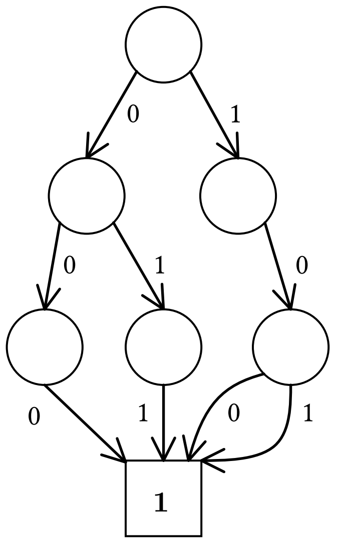
Definition 14 (Projection of state vectors) Given a set of variables V and a state vector s over these variables, s' is the projection of s to target variables $X \subseteq V$, denoted by $\mathrm{s} \searrow_X \mathrm{s}'$, iff $\mathrm{s}[k] = \mathrm{s}'[k]$ for every $x_k \in X$ and $\mathrm{s}'[l]$ is undefined for variables $x_l \notin X$ (i. e. $\mathrm{s}'[]$ is a partial function over V, assigning a value to elements in X only). A projection of a set of states S is denoted by $S \searrow_X S'$ where $S' = \{\mathrm{s}' \mid \exists \mathrm{s} \in S : \mathrm{s} \searrow_{V'} \mathrm{s}'\}$. ▪

Definition 15 (Projection of a PTS) A PTS $M' = (V', D, \mathcal{I}', \mathcal{E}, \mathcal{N}')$ is a projection of another PTS $M = (V, D, \mathcal{I}, \mathcal{E}, \mathcal{N})$, denoted by $M \searrow_{V'} M'$ if:

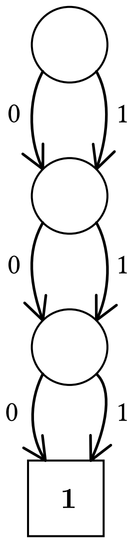
- $V' \subseteq V$ is the the set of target variables to which M is projected;

[4]The notation $s \subseteq s'$ suggests that s' represents more *real* states of the system and is consistent with the general notion of state abstraction and refinement.

[5]Notice the similarity with LTL model checking. In a sense, the Büchi automaton of an LTL property is a may abstraction of the described system.



(a) An MDD with 6 nodes encoding 4 states.



(b) An MDD with 3 nodes encoding 8 states.

Figure 2.3: Two MDDs illustrating how more encoded states can yield a smaller diagram. Nodes are displayed as circles, the terminal node as a box, while the *children* function is denoted by the labeled arcs between nodes. Nodes are assigned to variables by vertical layering of the layout, i. e. the lowest layer corresponds to decisions about x_1 while the root node on the top corresponds to x_3.

- $\mathcal{I} \searrow_{V'} \mathcal{I}'$, i. e. initial states are projected to the target variables;
- $\mathcal{N}' = \{(\mathrm{s}_1', \mathrm{s}_2') \mid \exists \mathrm{s}_1, \mathrm{s}_2 : (\mathrm{s}_1 \searrow_{V'} \mathrm{s}_1') \wedge (\mathrm{s}_2 \searrow_{V'} \mathrm{s}_2') \wedge (\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{N}\}$, i. e. there is a (projected) transition from projected state s_1' to s_2' iff there is a transition from any s_1 to any s_2 whose projections are s_1' and s_2', respectively.

When we speak about a single original PTS and a number of its projections, we will use the term *state* to refer to states of the original PTS and *partial state* over variables V' to refer to a state in a projection to V', denoted by $\mathrm{s}_{(V')}$ when it would be ambiguous. Special projections used throughout the dissertation are projections to $V_k = \{x_k\}$ (projection to the kth variable in the ordering), $V_{\leq k} = \{x_i \mid i \leq k\}$ (projection to first k variables in the ordering) and $V_{>k} = \{x_i \mid i > k\}$ (projection to last $K - k$ variables). ▪

2.3.3 Decision Diagrams

Saturation and symbolic model checking in general works with different types of decision diagrams. In this work, we consider multi-valued decision diagrams [MD98] to encode the state space of a PTS.[6] Two examples are shown in Figure 2.3.

Definition 16 (Multi-valued decision diagram) An ordered quasi-reduced multi-valued decision diagram (MDD) over a set of variables V ($|V| = K$), a variable ordering, and domains D is a tuple $MDD = (\mathcal{V}, lvl, children)$ where:

- $\mathcal{V} = \bigcup_{k=0}^{K} \mathcal{V}_k$ is the set of *nodes*, where items of $\mathcal{V}_0$ are the *terminal* nodes **1** and **0**, the rest ($\mathcal{V}_{>0} = \mathcal{V} \setminus \mathcal{V}_0$) are *internal* nodes ($\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ if $i \neq j$);
- $lvl : \mathcal{V} \to \{0, 1, \dots, K\}$ assigns non-negative *level numbers* to each node, associating them

[6]See [HTK08] for model checking with hierarchical set decision diagrams.

with variables according to the variable ordering (nodes in $\mathcal{V}_k = \{n \in \mathcal{V} \mid \mathit{lvl}(n) = k\}$ belong to variable x_k for $1 \leq k \leq K$ and are terminal nodes for $k = 0$);

- *children* : $\mathcal{V}_{>0} \times \mathbb{N} \to \mathcal{V}$ defines edges between nodes labeled with elements of $\mathbb{N}$ (denoted by $n[i] = \mathit{children}(n, i)$, $n[i]$ is left-associative), such that for each node $n \in \mathcal{V}_k$ ($k > 0$) and value $i \in D(x_k)$: $\mathit{lvl}(n[i]) = \mathit{lvl}(n) - 1$ or $n[i] = \mathbf{0}$;[7] as well as $n[i] = \mathbf{0}$ if $i \notin D(x_k)$;
- for every pair of nodes $n_1, n_2 \in \mathcal{V}_{>0}$, if for all $i \in \mathbb{N} : n_1[i] = n_2[i]$, then $n_1 = n_2$.

An MDD node $n \in \mathcal{V}_k$ encodes a set of partial states $S(n)$ over variables $V_{\leq k}$ such that for each $\mathbf{s} \in S(n)$ the value of $n[\mathbf{s}[k]] \cdots [\mathbf{s}[k]]$ (recursively indexing n with components of $\mathbf{s}$) is $\mathbf{1}$ and for all $\mathbf{s} \notin S(n)$ it is $\mathbf{0}$. When speaking about an MDD encoding the set $S(n)$, we mean the pair (MDD, n) and we refer to n as the root node of the MDD. The size of such an MDD is defined as the number of nodes reachable from the root node and is denoted by $|n|$. ▪

It is easy to see that indexing a node on level k in a decision diagram results in a node that encodes the states of a PTS projected to $V_{\leq k-1}$. In this sense, terminal nodes belong to a projection with no variables at all, which has a single state (ε, a zero-length vector) that is either included in $S(n)$ (in this case $n = \mathbf{1}$ or not ($n = \mathbf{0}$). Furthermore, the encoded set of states of a node can be expressed in terms of its children: $S(n) = \bigcup_{i \in \mathbb{N}} \left(\{i\} \times S(n[i])\right)$. For the sake of convenience, we will use $S_{[n]}(i) = \{i\} \times S(n[i])$ to denote the partition of $S(n)$ where $x_k = i$ (note that these partitions are disjoint).

Based on this idea, there are efficient recursive algorithms that compute the result of set operations directly on MDDs (e. g. union is described in [CMS06]). They use the following equivalence to decompose the computation recursively, caching the results on each level to reuse them when the same nodes are reached on different paths:

$$S(n_1)\ \mathit{op}\ S(n_2) \equiv \bigcup_{i \in \mathbb{N}} \Big(\{i\} \times \big(S(n_1[i])\ \mathit{op}\ S(n_2[i])\big)\Big) \tag{2.1}$$

Recursion stops at terminal nodes, where the result is easily computed because only two sets are possible: $\{\varepsilon\}$ and $\varnothing$. With caching, each pair of nodes is processes only once, so these algorithms run in $\mathcal{O}(|n_1| \cdot |n_2|)$ steps assuming that the number of arcs from each node can be bound by a constant (which is practically the case with finite domains).

An interesting property of MDDs is that the number of nodes does not grow proportionally with the size of the encoded set. In fact, the size of an MDD can decrease when adding new states because of the exploited regularities. This phenomenon can be observed on Figure 2.3, where the MDD in Figure 2.3a encodes 4 states with 6 internal nodes, while the one in Figure 2.3b encodes 8 states with 3 internal nodes.

2.3.4 Next-State Representations

If sets of states are encoded in decision diagrams, it is advised to encode the next-state relation in a similar way. Operations on decision diagrams are efficient only with recursion and caching, otherwise the encoded elements had to be considered one by one, like in simple set operations. Therefore, we prefer next-state representations where the *relational product* operation (see Section 2.3.5) can be defined recursively.

[7] Quasi-reduced means that arcs may skip levels only if they lead to $\mathbf{0}$, with the meaning that regardless of the values of skipped variables the result will be the same. With the same meaning, fully reduced decision diagrams allow this even when the target is not $\mathbf{0}$. A fully reduced decision diagram can be interpreted as quasi-reduced if the *children* function gets the current level number as well and returns the same node regardless of the index if the node is not on the current level.

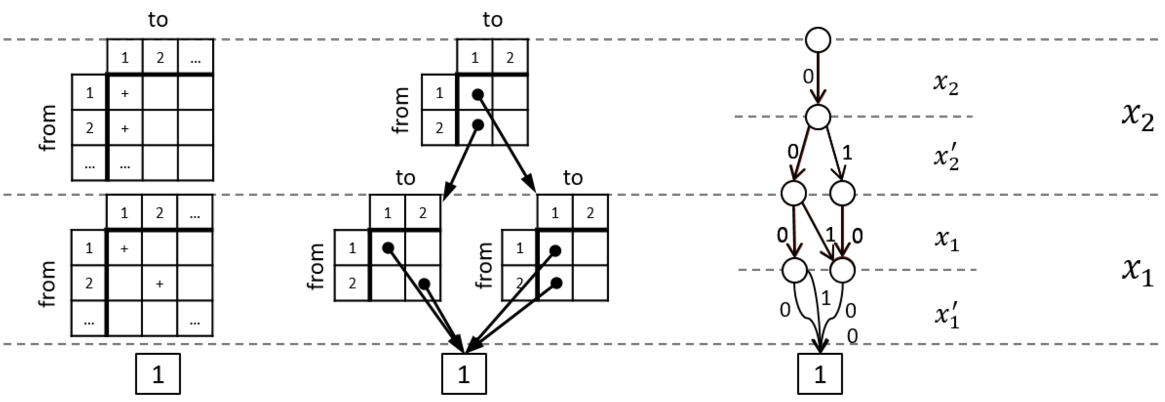


Figure 2.4: Examples for three kinds of next-state representations (from left to right: Kroenecker matrices, Matrix Diagrams and 2K-MDDs).

2.3.4.1 Non-Recursive Representations.

Transitions are essentially directed edges of the state graph, therefore two natural ways to represent them are adjacency lists and adjacency matrices. An adjacency list assigns to each (source) state a list of target states, while an adjacency matrix is a matrix $\mathbf{A} = (a_{ij}) \in \mathbb{B}^{|\mathcal{S}|\times|\mathcal{S}|}$ where each a_{ij} is true if $(\mathbf{s}_i, \mathbf{s}_j) \in \mathcal{N}$ and false otherwise. These representations work with states as atomic elements instead of state vectors, therefore they are not suitable for recursive processing similar to decision diagrams.

2.3.4.2 Kroenecker Consistent Representations

If states are vectors, the *Kroenecker product* operation on matrices might be used to decompose the system. This is not always the case: the transition relation has to have specific symmetries. The Kroenecker product of matrix $\mathbf{A}$ with dimensions $m \times n$ and $\mathbf{B}$ with dimensions $p \times q$ is the $mp \times nq$ block matrix defined as follows ($a\mathbf{B}$ is element-wise conjunction with a Boolean literal in this case):

$$\mathbf{A} \oplus \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix} \tag{2.2}$$

Definition 17 (Kroenecker consistency of a PTS) A next-state relation of a PTS $\mathcal{N}$ (or a partition thereof) is *Kroenecker consistent* iff for each $x_k \in V$ there exists an adjacency matrix $\mathbf{A}_k$ with dimensions $|D(x_k)| \times |D(x_k)|$ such that the adjacency matrix $\mathbf{A}$ describing $\mathcal{N}$ is $\mathbf{A}_1 \oplus \cdots \oplus \mathbf{A}_K$. ▪

It is interesting to note that $\mathbf{A}_i$ describes the next-state relation of a projection of the PTS to V_k. The series of these projections is called a *Kroenecker matrix* representation and can be indexed recursively to check if a transition $(\mathbf{s}, \mathbf{s}')$ is in $\mathcal{N}$: starting from $\mathbf{A}_K$, if $\mathbf{A}_k\big[\mathbf{s}[k], \mathbf{s}'[k]\big]$ is false then the transition is not in $\mathcal{N}$, otherwise check $\mathbf{A}_{k-1}$ unless $k = 1$, in which case the transition is in $\mathcal{N}$. This strategy implies a linked list structure where matrices are linked according to the variable order. The left-most part of Figure 2.4 illustrates the concept where the encoded transition relation sets x_2 to 1 and leaves x_1 unchanged.

Since a next-state relation consisting of a single transition $\mathcal{N} = \{(\mathbf{s}, \mathbf{s}')\}$ is always Kroenecker consistent, there is a partitioning for every next-state relation where every partition is Kroenecker consistent. A high-level characterization of a Kroenecker consistent next-state relation is that *1)* the next value of a variable can depend only on the current value of the same variable and *2)* each variable has a set of *enabling* values, such that the transition is enabled if the values of all variables enable it. The next-state relation of any Petri net (without priorities) is partitioned by its transitions into subsets that are Kroenecker consistent.[8]

Even in Kroenecker consistent models, next-state relations are rarely given explicitly in Kroenecker matrix form. More often they are defined implicitly by partial functions for each variable, which are defined over subsets of the variable's domain and compute the next value from the current one. This most general representation is used in [BM19] where they call it *implicit relation forests*, referring to the fact that for a set of next-state relations, identical suffixes of both Kroenecker matrix chains and implicit relation chains can be merged for compactness and better caching.

For Petri nets, a special case of implicit relations can be given with only 3 integer values for each space: the number of tokens that *enable* (due to simple arcs) and *disable* (due to inhibitor arcs) the Petri net transition as well as the offset in the next marking (weight difference of output and input arcs).

2.3.4.3 Decision Diagram Representations

Even though every next-state relation can be decomposed into Kroenecker consistent partitions, this is generally expensive if the high-level transitions do not already define such a partitioning (such as in Petri nets). For such cases, different types of decision diagrams have been introduced.

From the MDD point of view, a natural idea is to use twice the number of variables to encode the next-state relation, where simple variables x_k correspond to current values and primed variables x'_k to next values [CMS03]. Conventionally, the variable ordering in such a diagram is $x_1, x'_1, \dots, x_K, x'_K$.[9] We will refer to this approach as 2K-MDD.[10] The right-most part of Figure 2.4 illustrates the concept where the encoded transition relation is the following: $(00 \to 00), (00 \to 10), (00 \to 01), (10 \to 00)$, the order of values is $x_1 x_2$.

From the Kroenecker matrix point of view, another representation with the same expressive power is called *matrix diagram* [Min01]. Matrix diagrams can be regarded as a hybrid of MDDs and Kroenecker matrices: instead of matrix chains, we organize matrices into a diagram, allowing cells to not only specify enabledness (as a Boolean value), but also the next matrix on the lower level.

Definition 18 (Matrix diagram) An ordered quasi-reduced matrix diagram (MxD) over a set of variables V ($|V| = K$), a variable ordering, and domains D is a tuple $MDD = (\mathcal{V}, lvl, matrix)$ where:

- $\mathcal{V} = \bigcup_{k=0}^{K} \mathcal{V}_k$ is the set of *nodes*, where items of $\mathcal{V}_0$ are the *terminal* nodes **1** and **0**, the rest ($\mathcal{V}_{>0} = \mathcal{V} \setminus \mathcal{V}_0$) are *internal* nodes ($\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ if $i \neq j$);
- $lvl \colon \mathcal{V} \to \{0, 1, \dots, K\}$ assigns non-negative *level numbers* to each node, associating them with variables according to the variable ordering (nodes in $\mathcal{V}_k = \{m \in \mathcal{V} \mid lvl(m) = k\}$

[8]This is one of the reasons it is worth including events in the definition of PTSs.

[9]There are various reasons to follow this convention, e. g. the next value is most of the times related to the current value, resulting in a more compact diagram.

[10]There is an important difference between *fully reduced* MDDs and 2K-MDDs. While a skipped level in an MDD means that the variable can have any value, this applies only to even levels of 2K-MDDs (corresponding to simple variables x_k). If an odd level is skipped (corresponding to primed variables x_k), it means that the only index not associated with the **0** node is the same as the index of the level skipping arc. This implies that *identity* relations can be reduced, yielding a constant-sized representation for identities instead of the linear-sized enumeration of every identical pair.

belong to variable x_k for $1 \leq k \leq K$ and are terminal nodes for $k = 0$);

- *matrix* $= \bigcup_{1 \leq k \leq K}$ *matrix*$_k$ with *matrix*$_k : \mathcal{V}_k \to \mathcal{V}_{<k}^{|D(x_k)| \times |D(x_k)|}$ defines the *edge matrix* of a node (indexing is denoted by $m[i,j] = \mathit{matrix}(m)_{ij}$, $n[i,j]$ is left-associative), such that for each node $m \in \mathcal{V}_k$ ($k > 0$) and values $i, j \in D(x_k) : \mathit{lvl}(m[i,j]) = \mathit{lvl}(m) - 1$ or $m[i,j] = \mathbf{0}$;[11]as well as $m[i,j] = \mathbf{0}$ if $i \notin D(x_k)$ or $j \notin D(x_k)$;
- for every pair of nodes $m_1, m_2 \in \mathcal{V}_{>0}$, if for all $i, j \in \mathbb{N} : m_1[i,j] = m_2[i,j]$, then $m_1 = m_2$.

An MxD node $m \in \mathcal{V}_k$ encodes a relation $N(m)$ between partial states over variables $V_{\leq k}$ such that for each $(\mathbf{s}, \mathbf{s}') \in N(m)$ the value of $m[\mathbf{s}[k], \mathbf{s}'[k]] \cdots [\mathbf{s}[k], \mathbf{s}'[k]]$ (recursively indexing m with components of $\mathbf{s}$ and $\mathbf{s}'$) is $\mathbf{1}$ and for all $(\mathbf{s}, \mathbf{s}') \notin N(m)$ it is $\mathbf{0}$. When speaking about an MxD encoding the set $N(m)$, we mean the pair (MxD, m) and we refer to m as the root node of the MxD. ▪

Matrix diagrams are a very intuitive way to represent a next-state relation. They follow the structure of MDDs, but each node has to be indexed with two values. The indexing of a node corresponds to the change of a single variable x_k and recursively yields another node describing what can happen to the other variables in $V_{<k}$ given they way x_k would change. The middle part of Figure 2.4 illustrates the concept where the encoded transition relation sets x_2 to 1, and if x_2 was originally 2, it also sets x_1 to 1 (otherwise it is unchanged).

Before the introduction of *Abstract Next-State Diagrams* (ANSD) in Thesis 3 (Chapter 6), we choose to present algorithms with the matrix diagram representation.

2.3.5 State Space Generation with Set Operations

With compact representations for sets and relations, as well as efficient set operations, the last missing element for symbolic model checking is the recursive definition of the relational product operation. Formally, the relational product of a (state) set $\mathcal{S}$ and a (next-state) relation $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is $\mathcal{S} \circ \mathcal{N} = \mathcal{N}(\mathcal{S}) = \{\mathbf{s}' \mid \exists \mathbf{s} \in \mathcal{S} : (\mathbf{s}, \mathbf{s}') \in \mathcal{N}\}$. With a decision diagram node n and a matrix diagram node m ($\mathit{lvl}(n) = \mathit{lvl}(m)$, same set of variables and same variable order), the recursive computation of the relational product $S(n) \circ N(m)$ is based on the following equivalence:

$$S(n) \circ N(m) \equiv \bigcup_{j \in \mathbb{N}} \{j\} \times \Big(\bigcup_{i \in \mathbb{N}} S(n[i]) \circ N(m[i,j]) \Big) \tag{2.3}$$

Recursion stops at terminal nodes, where the results are easily computed based on $S(\mathbf{1}) = \{\varepsilon\}$, $S(\mathbf{0}) = \emptyset$, $N(\mathbf{1}) = \{(\varepsilon, \varepsilon)\}$ (the terminal identity relation is a 1×1 adjacency matrix with $a_{11} = \mathit{true}$) and $N(\mathbf{0}) = \emptyset$ (the terminal empty relation is 1×1 adjacency matrix with $a_{11} = \mathit{false}$).

As mentioned in Section 2.1.3, computing the set of reachable states of a PTS is equivalent to computing $\mathcal{I} \circ \mathcal{N}^* = \mathcal{N}^*(\mathcal{I}) = \mathcal{I} \cup \mathcal{N}(\mathcal{I}) \cup \mathcal{N}\big(\mathcal{N}(\mathcal{I})\big) \cup \cdots$, where $\mathcal{N}^*$ is the reflexive transitive closure of the next-state relation. Another way to express this – and many other model checking problems – is using fixed points.

A *fixed point* of a function $f : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ is a subset of the base set $S \subseteq \mathcal{S}$ such that $S = f(S)$. For *monotonically increasing* functions (i. e. $S \subseteq f(S)$), a *least fixed point* is a fixed point S such that $S \subseteq S'$ for every fixed point S'. Similarly, for *monotonically decreasing* functions (i. e. $S \supseteq f(S)$), a *greatest fixed point* is a fixed point such that $S \supseteq S'$ for every fixed point S'. We will often require least fixed points to contain as subset a set of (initial) states $\mathcal{I}$, which can be expressed as the least

[11]A *fully reduced* matrix diagram allows skipping a level if the matrix of the reduced node would be an identity matrix (with diagonal elements pointing to the same child node and non-diagonal elements pointing to the $\mathbf{0}$ node). This is analogous to the reduction rules of 2K-MDDs.

fixed point of $f'(S) = f(S) \cup \mathcal{I}$, or greatest fixed points to be a subset of a set of (reachable) states $\mathcal{S}_r$, expressed as the greatest fixed point of $f'(S) = f(S) \cap \mathcal{S}_r$.

Expressed as a fixed point, the set of reachable states is a least fixed point $\mathcal{S}_r = \mathcal{I} \cup \mathcal{S}_r \cup \mathcal{N}(\mathcal{S}_r)$ (with $f(\mathcal{S}_r) = \mathcal{S}_r \cup \mathcal{N}(\mathcal{S}_r)$ being a monotonically increasing function and $\mathcal{I} \cup f(\mathcal{S}_r)$ enforcing the containment of initial states).

A finite fixed point of monotonic function f over a finite base set can be computed iteratively by starting from the bottom element (e. g. the empty set $\varnothing$) for least fixed points, the top element (e. g. the potential state space $\hat{S}$) for greatest fixed points, or an arbitrary element for any fixed point, and repeatedly applying f until $f(S) = S$ becomes true. This scheme yields the strategy to compute the set of reachable states as $\mathcal{N}^*(\mathcal{I}) = \mathcal{I} \cup \mathcal{N}(\mathcal{I}) \cup \mathcal{N}\big(\mathcal{N}(\mathcal{I})\big) \cup \cdots$, which is the result of recursively expanding the fixed point as if it was an assignment (instead of an equality).

The strategy above is equivalent to the breadth-first search (BFS) algorithm for graph traversal. For certain types of systems expressed as PTSs, BFS with *chaining* was shown to work better [CCY06]. Chaining exploits the partitioning of the next-state relation and instead of applying the whole relation in each step, it chains the application of partitions (i. e. the firing of high-level events) to progress faster. The definition of f for BFS with a partitioned next-state relation applies each partition to the parameter set as follows:

$$f(S) = S \cup \big(\mathcal{N}_{e_1}(S) \cup \cdots \cup \mathcal{N}_{e_{|\mathcal{E}|}}(S)\big) \tag{2.4}$$

In contrast, BFS with chaining defines f by applying next-state partitions to all discovered states:

$$f(S) = S \cup \mathcal{N}_{e_1}(S) \cup \mathcal{N}_{e_2}\big(S \cup \mathcal{N}_{e_1}(S)\big) \cup \cdots \cup \mathcal{N}_{e_{|\mathcal{E}|}}\big(S \cup \mathcal{N}_{e_{|\mathcal{E}|-1}}(\ldots)\big) \tag{2.5}$$

Of course, parameters of the next-state relation partitions are not computed again, but reused from left-to-right computation of the whole expression. Another version of chaining uses the reflexive transitive closures $\mathcal{N}_e^*$ of $\mathcal{N}_e$, which can be computed as another least fixed point.

The next section presents the saturation algorithm, which is an even more natural and efficient approach for PTSs.

2.4 Saturation

The problem with BFS is that firing all events from a set of states yields a lot of different state vectors, whereas decision diagrams are compact for sets of similar vectors. This often results in very large intermediate MDDs, even if the set of reachable states does have a compact MDD representation. On top of the implied memory overhead, execution time also increases drastically as most MDD operations scale in the size of their operands. Chaining helps because it fires events from all previously explored states, yielding more similar state vectors and a more compact MDD representation for the intermediate results.

Saturation puts the emphasis on compactness of intermediate decision diagrams and uses a strategy that guides the exploration accordingly. This section will introduce the concept of locality (Section 2.4.1) and the saturation algorithm (Section 2.4.2), as well as a variant called constrained saturation (Section 2.4.3), which introduces key ideas with regard to the research presented in this work.

2.4.1 Locality

Exploiting the information preserved in a PTS, we can define different relationships between an event and a variable (illustrated in the dependency matrices of Figure 2.5).

	cons.	msg.	prod.
t_1			rw
t_2		rw	rw
t_3		rw	
t_4	rw	rw	
t_5	rw		

(a) Simple version.

	cons.	msg.	prod.
t_1			rw
t_2		rw	rw
t_3		rw	**r**
t_4	rw	rw	
t_5	rw		

(b) With inhibitor arc.

	cons.	msg.	prod.
t_1	**r**	**r**	rw
t_2		rw	rw
t_3	**r**	rw	**r**
t_4	rw	rw	
t_5	rw	**r**	**r**

(c) With timed transitions.

Figure 2.5: Dependency matrices for the 3 variants of the example. Letters *r* and *w* stand for *read* and/or *write* dependencies, differences from the simple model are in boldface. Colors denote the highest component that is not independent from the transitions, an information used by the saturation algorithm.

Definition 19 (Locally read-only) An event e is *locally read-only* on variable x_k if for any $(\mathrm{s}, \mathrm{s}') \in \mathcal{N}_e$ we have that $\mathrm{s}[k] = \mathrm{s}'[k]$. Informally, the value of x is never modified by the transitions of event e. ▪

While the locally read-only property guarantees that the value of the variable will not change, the event can still depend on the information stored in the variable. The following property forbids this as well.

Definition 20 (Locally invariant) An event e is *locally invariant* on variable x_k if it is locally read-only and for any $(\mathrm{s}, \mathrm{s}') \in \mathcal{N}_e$ and $v \in D(x_k)$ we also have $(\mathrm{s}_{[x_k \leftarrow v]}, \mathrm{s}'_{[x_k \leftarrow v]}) \in \mathcal{N}_e$, where $\mathrm{s}_{[x_k \leftarrow v]}$ is a state where the value of variable x_k is v, but all other variables have the same value as in s. Informally, the value of x does not affect the outcome of event e. ▪

With the help of local invariance, we can define locality, the central notion of the saturation algorithm.

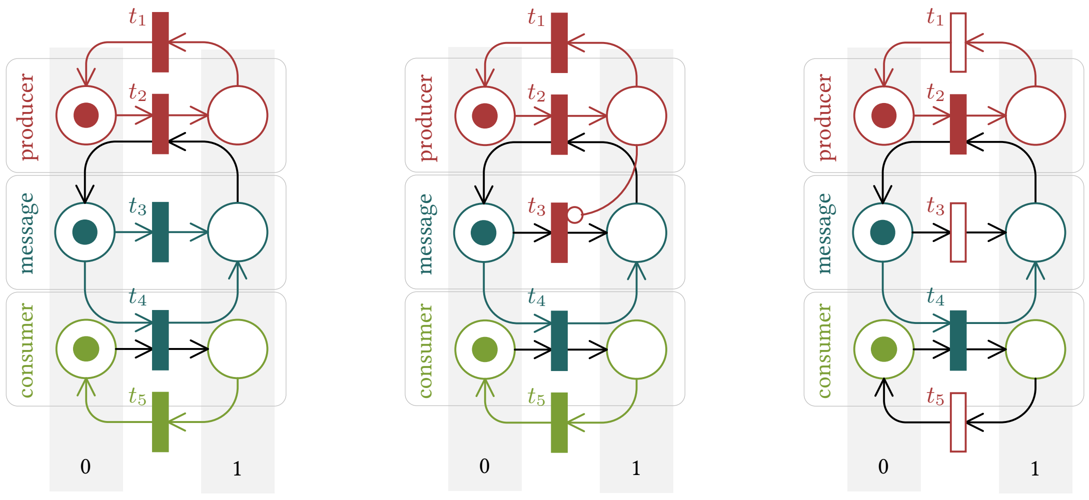
Definition 21 (Locality) An event $e \in \mathcal{E}$ is said to be *local* over variables $X \subseteq V$ if it is locally invariant on variables in $V \setminus X$. If X is minimal (i. e. the event is dependent on variables in X) then we say that X is the set of *supporting variables* of e: $Supp(e) = X$. With respect to the variable order, it is practical to distinguish the variable with the highest index among the supporting variables, which is the *top* variable ($Top(e)$) of e, and that with the lowest index, called the *bottom* variable ($Bot(e)$). In the context of saturation, we will use $\mathcal{E}_k = \{e \mid Top(e) = x_k\}$ and $\mathcal{N}_k = \bigcup_{e \in \mathcal{E}_k} \mathcal{N}_e$ to denote events and their next-state relations whose top variable is x_k. ▪

The next-state relation of an event e local on variables $Supp(e) = X$ can be defined over partial states $S_{(X)}$, because no other information is required to compute its image. This enables a compact representation[12] and clever iteration strategies like saturation.

2.4.2 The Saturation Algorithm

Saturation is a symbolic state space generation algorithm working on decision diagrams [CMS06]. Formally, given a PTS M, its goal is to compute the set of states $\mathcal{S}_r$ that are reachable from the initial states $\mathcal{I}$ through transitions in $\mathcal{N}$: $S = \mathcal{I} \cup \mathcal{N}(\mathcal{I}) \cup \mathcal{N}\big(\mathcal{N}(\mathcal{I})\big) \cdots = \mathcal{N}^*(\mathcal{I})$, where $\mathcal{N}^*$ is the reflexive

[12]In fully reduced 2K-MDDs and MxDs, there will be no node on the levels corresponding to independent variables, because they would be identities and therefore reduced.



(a) Simple version. (b) With inhibitor arc. (c) With timed transitions.

Figure 2.6: Submodels for saturation corresponding to each component. Colors denote the assignment of transitions to submodels. Arcs that are responsible for the assignment (introducing the dependency to the higher level) are also colored. Note that dependencies introduced by priorities are not visible (see Figure 2.5 for the dependency matrices).

and transitive closure of $\mathcal{N}$. This is equivalent to computing the least fixed point characterized as the minimal solution of $S = \mathcal{I} \cup S \cup \mathcal{N}(S)$. The main strength of saturation is a recursive computation of this fixed point, which is based on the following definitions.

Based on locality, first define the *submodel* of a PTS, which is similar to a projection, but analogous to must abstractions.

Definition 22 (Submodel of a PTS) Given a PTS $M = (V, D, \mathcal{I}, \mathcal{E}, \mathcal{N})$ and a set of target variables $V' \subset V$, another PTS $M' = (V', D, \mathcal{I}', \mathcal{E}', \mathcal{N}')$ is a *submodel* of M implied by V' (denoted by $M \succeq_{V'} M'$) if:

- $\mathcal{I} \searrow_{V'} \mathcal{I}'$, i. e. initial states are projected to the target variables;
- $\mathcal{E}' = \{e \mid \mathit{Supp}(e) \subseteq V'\}$, i. e. events are restricted to those of which are local over the set of target variables;
- $\mathcal{N}' = \bigcup_{e \in \mathcal{E}'} \mathcal{N}'_e$, where $\mathcal{N}'_e$ is the *projection* of the next-state relations corresponding to the local events in the original PTS M (like in Definition 15), i. e. only local transitions are kept. ▪

A submodel is a must abstraction, because only local events are included, which will be fireable regardless of the omitted variables. Submodels can be used to divide the state space exploration problem. To introduce the terminology of saturation gradually, we start with the definition of saturated sets of states, which represent *local fixed points* during state space exploration.

Definition 23 (Saturated set of states) Given a PTS M, a set of variables V' and the submodel $M \succeq_{V'} M'$ implied by the variables, a set of (partial) states S over variables V' is *saturated* iff $S = S \cup \mathcal{N}'(S)$, where $\mathcal{N}'$ is the next-state relation of the submodel. ▪

Combining local fixed points on submodels with decision diagrams yields the definition of a saturated node. We will use submodels over an increasing prefix of the variable ordering, i. e.

$\varnothing, V_{\leq 1}, V_{\leq 2}, ...$, denoted by $M_{\varnothing}, M_{\leq 1}, M_{\leq 2}, ...$, respectively. We will use the same notation to refer to events and next-state relations of $M_{\leq k}$ (like $\mathcal{E}_{\leq k}$ and $\mathcal{N}_{\leq k}$).

> **Definition 24 (Saturated node)** Given a PTS M, an MDD node n on level $lvl(n) = k$ is *saturated* iff it encodes a set of (partial) states $S(n)$ that is saturated with respect to the submodel corresponding to $V_{\leq k}$. Equivalently, the node is saturated iff all of its children $n[i]$ are saturated and $S(n) = S(n) \cup \mathcal{N}_k(S(n))$. ▪

The two definitions are equivalent because of the following. In a fixed point, no transition can reach "new" states, so the fixed point over variables $V_{\leq k}$ implies that no transition in $\mathcal{N}_{\leq k}$ will leave $S(n)$. Similarly, the fixed point $S(n) = S(n) \cup \mathcal{N}_k(S(n))$ implies that no transition in $\mathcal{N}_k$ will leave $S(n)$. In the recursive step, we have that for each child $n[i]$, transitions in $\mathcal{N}_{\leq k-1}$ will not leave $S_{[i]}(n)$. Using the recursive definition of $S(n)$ we can derive that transitions in $\mathcal{N}_{\leq k-1}$ will not leave $S(n)$. Since $\mathcal{N}_{\leq k} = \mathcal{N}_k \cup \mathcal{N}_{\leq k-1}$, we proved that the two definitions do in fact define the same fixed point.

As suggested by the definition, locality is mainly used to compute a *Top* value for each event, which is the lowest level on which fixed point computation involving the event can happen. By definition, the terminal nodes **1** and **0** are saturated because they do not have child nodes and $\mathcal{N}_{\varnothing}$ (the next-state relation of the empty submodel) is empty. The saturation algorithm is then easily defined as a recursive algorithm that given a node n computes the *least* fixed point $S(n_s) = S(n_s) \cup \mathcal{N}_k(S(n_s))$ that contains $S(n)$, making sure that child nodes are always saturated by recursion. Figure 2.6 illustrates which events are processed on which level on the example Petri nets from Figure 2.1. When applied on a node encoding the set of initial states, the result will be a node encoding the states reachable through transitions in $\mathcal{N}$.

The motivation of this decision diagram-driven strategy comes from the observation that larger sets may often be encoded in smaller MDDs. By exploring as many variations in the lower variables as possible, intermediate diagrams may be much smaller than in traditional BFS and chaining BFS strategies (also described in [CMS06]). Another intuition is that in an MDD encoding the set of reachable states, all nodes are by definition saturated – therefore it is impractical to create nodes which have unsaturated child nodes. In other words, a saturated node has a chance of being in the final MDD, while an unsaturated one does not.

It is interesting to note that saturation was believed to be hard to parallelize [CZJ09]. Results of [Vör+11] and in particular the solution implemented in the LTSmin[13] tool [DMP19], however, introduced more and more efficient parallelization strategies, reaching nearly-linear scaling with the number of processing cores. Both of them relies on parallel processing of nodes, where the processing of child nodes start in parallel with the parent. These approaches are orthogonal to the algorithms presented in this dissertation, as every operation of every proposed algorithm follows the same node-wise processing as the original saturation algorithm, essentially yielding the same task hierarchy.

2.4.3 The Constrained Saturation Algorithm

The *constrained saturation algorithm* has been introduced in [ZC09] to limit the exploration inside the boundaries of a predefined set of states (the constraint). Even though this is possible with the original algorithm by removing transitions in $\mathcal{N}$ that end in states not inside the constraint, it would damage the locality of events by making them dependent on additional variables (the event has to decide whether it is leaving the constraint or not). Constrained saturation avoids this by traversing

[13] https://ltsmin.utwente.nl/

an MDD representation of the constraint along with the MDD of the state space, and deciding the enabledness of events when firing them.

Formally, given a constraint set C, the goal of constrained saturation is to compute the least fixed point $S = S \cup \big(\mathcal{N}(S) \cap C\big)$ that contains the initial states inside the constraint $\mathcal{I} \cap C$. Definitions 23 and 24 are modified as follows.

Definition 25 (Saturated state space with constraint) Given a PTS M, a set of variables V', the submodel $M \trianglerighteq_{V'} M'$ implied by the variables and a constraint C, a set of (partial) states S over variables V' is *saturated* iff $S = S \cup \big(\mathcal{N}'(S) \cap C\big)$, where $\mathcal{N}'$ is the next-state relation of the submodel. ▪

Definition 26 (Saturated node with constraint) Given a PTS M and a constraint node n_c ($S(n_c) = C$), an MDD node n on level $lvl(n) = k$ is *saturated* iff it encodes a set of (partial) states $S(n)$ that is saturated with respect to the submodel corresponding to $V_{\leq k}$ and the constraint C. Equivalently, the node is saturated iff all of its children $n[i]$ are saturated with respect to constraint node $n_c[i]$ and $S(n) = S(n) \cup \big(\mathcal{N}_k(S(n)) \cap S(n_c)\big)$, where $\mathcal{N}_k = \bigcup_{e \mid Top(e) = x_k} \mathcal{N}_e$ for $1 \leq k \leq K$ and $\mathcal{N}_0 = \emptyset$. ▪

The recursive computation of $\mathcal{N}_k(S(n)) \cap S(n_c)$ is done by simultaneously traversing n with the source states, a recursive representation of $\mathcal{N}_k$ with source and target states, and n_c with target states. Note that n_c does not encode the partial state determined by the path through which recursion reached the current node, but "remembers" just enough to decide if the transition is allowed based only on the rest of the state.

Figures 2.7a–2.7c present the pseudocode of the constrained saturation algorithm. To retrieve the pseudocode of the original saturation algorithm, one should assume that at any point $c \neq \mathbf{0}$ and $c[i] \neq \mathbf{0}$ for any i. The pseudocode also contains a stub for the Confirm procedure that serves for the on-the-fly update of the next-state relations whenever new states are found (as described in [CMS06] and enhanced in [Mei+14]).

The ConsSaturate procedure starts by checking the terminal cases. Line 2 checks if the same problem has already been solved. Caching – as in all operations on decision diagrams – is crucial to have optimal performance. If there is no matching entry in the cache, the algorithm recursively saturates the children of the input node n, calling Confirm for every encountered local state. The resulting node is checked for uniqueness in line 8 and is replaced by an already existing node if necessary (to preserve MDD canonicity). In line 11, we iterate over the MxD nodes for each event belonging to the current level, apply them again and again in lines 9–13 until no more states are discovered – a fixed point is reached. This version of the iteration is called *chaining* and is discussed in [CMS06].

The result of firing an event on a set of states is computed by ConsFire and ConsRecFire. The only differences between them are that ConsRecFire also saturates the resulting node before returning it and also caches it – ConsFire is called as part of a saturation process so this is not necessary. The common parts (3–7 in ConsFire and 4–8 in ConsRecFire) compute the resulting node by recursively processing their child nodes. It is important to note that the arguments of the recursive call are $n[i]$, $c[j]$ and $m[i, j]$, that is, n is traversed along the source state and c is traversed along the target state. The recursive saturation of the result node in ConsRecFire in line 10 ensures that child nodes of the currently saturated node always stay saturated during the fixed point computation in accordance with Definitions 24 and 26.

```
Input: MDD node $n$, $c$
Output: saturated MDD node $n'$
1  if $n = \mathbf{0}$ or $n = \mathbf{1}$ or $c = \mathbf{0}$ then return $n$
2  if $\neg$SatCacheGet$(n, c, n')$ then
3      $n' \leftarrow$ new MddNode$(lvl(n))$
4      for each $i$ where $n[i] \neq \mathbf{0}$ do
5          Confirm$(lvl(n), i)$
6          $n'[i] \leftarrow$ ConsSaturate$(n[i], c[i])$
7      end
8      $n' \leftarrow$ CheckIn$(n')$
9      repeat
10         changed $\leftarrow$ false
11         for each $m \in \{m \mid \exists e \in \mathcal{E}_{lvl(n)} : N(m) = \mathcal{N}_e\}$ do
12             $n'' \leftarrow$ ConsFire$(n', c, m)$
13             if $n' \neq n''$ then $n' \leftarrow n''$, changed $\leftarrow$ true
14         end
15     until $\neg$changed
16     SatCachePut$(n, m, n')$
17 end
18 return $n'$
```

(a) Procedure ConsSaturate.

```
Input: MDD node $n$, $c$, MxD node $m$
Output: MDD node $n'$ encoding states in $S(c)$ reachable from $S(n)$ through $N(m)$ such that children of $n'$ are
        saturated
1  if $n = \mathbf{0}$ or $m = \mathbf{0}$ then return $\mathbf{0}$
2  if $m = \mathbf{1}$ then return $n$
3  $n' \leftarrow$ new MddNode$(lvl(n))$
4  for each $i, j$ where $n[i] \neq \mathbf{0}$ and $c[j] \neq \mathbf{0}$ and $m[i, j] \neq \mathbf{0}$ do
5      $s \leftarrow$ ConsRecFire$(n[i], c[j], m[i, j])$
6      if $s \neq \mathbf{0}$ then Confirm$(lvl(n), j)$
7      $n'[j] \leftarrow n'[j] \cup s$
8  end
9  $n' \leftarrow$ CheckIn$(n')$
10 return $n' \cup n$
```

(b) Procedure ConsFire.

```
Input: MDD node $n$, $c$, MxD node $m$
Output: MDD node $n'$ encoding states in $S(c)$ reachable from $S(n)$ through $N(m)$ such $n'$ is saturated
1  if $n = \mathbf{0}$ or $m = \mathbf{0}$ then return $\mathbf{0}$
2  if $m = \mathbf{1}$ then return $n \cap c$
3  if $\neg$RecFireCacheGet$(n, c, m, n'')$ then
4      $n' \leftarrow$ new MddNode$(lvl(n))$
5      for each $i, j$ where $n[i] \neq \mathbf{0}$ and $c[j] \neq \mathbf{0}$ and $m[i, j] \neq \mathbf{0}$ do
6          $s \leftarrow$ ConsRecFire$(n[i], c[j], m[i, j])$
7          if $s \neq \mathbf{0}$ then Confirm$(lvl(n), j)$
8          $n'[j] \leftarrow n'[j] \cup s$
9      end
10     $n' \leftarrow$ CheckIn$(n')$, $n'' \leftarrow$ ConsSaturate$(n', m)$, RecFireCachePut$(n, c, m, n'')$
11 end
12 return $n''$
```

(c) Procedure ConsRecFire.

Figure 2.7: Pseudocode of constrained saturation with MxDs.

Chapter 3

Analysis of Generalized Stochastic Petri Nets with Symbolic State Space Generation

3.1 Introduction

Priorities in Petri nets provide a convenient way to represent dependencies between transitions, making them useful in the modelling of complex problems. One particularly important subset of prioritized Petri nets is Generalized Stochastic Petri nets (GSPN, introduced in Section 2.1.2). To analyse the stochastic behaviour of a GSPN, the model must not express any nondeterminism. One way to guarantee this is to assign priorities to the transitions [TFP03]. While explicit (graph-based) model checking algorithms naturally handle priorities, symbolic model checkers often have trouble representing the resulting complex transition relations compactly.

Furthermore, locality of prioritized transitions are determined not only by read and write dependencies, but also other transitions with higher priority. In general, to determine the fireability of a transition in a prioritized Petri nets, we need to check the enabledness of all transitions with higher priority, introducing a read dependency for every place used by these higher-priority transitions.

The problem of representation can be overcome with decision diagram-like representations (see Section 2.3.4), but the problem of reduced locality remains. The approach in [Min04] uses matrix diagrams to encode priorities into the transition relations of Petri nets by removing elements where the source state enables a higher-priority transition. To compensate for the degraded locality, they present a method to factor the relations (i. e. merging events and decomposing them in a way independent from the original model) such that saturation can still exploit some of the original locality.

The motivation of this thesis comes from the intuition that any alteration to the transition relations (without priorities) that affects locality will hurt the efficiency of saturation more than what is absolutely necessary. Therefore we devised a solution that, with the modification of the saturation algorithm (inspired by constrained saturation, presented in Section 2.4.3), uses the simple next-state relations of unprioritized transitions as is and handles the priority-related fireability separately, encoded in a new kind of decision diagram called *edge-valued interval decision diagram* (EVIDD). We show that for Petri nets, such a diagram can be constructed offline.

We expect our approach to yield smaller intermediate decision diagrams and thus result in better performance for the state space generation of prioritized models. Our experiments comparing our results to that of [Min04] confirm this expectation, demonstrating that the presented algorithm scales better with the size of benchmark models than previous approaches.

The chapter is structured as follows. In the rest of this section, we provide an overview of the problem to be solved, including alternatives and related work (Section 3.1.1). Section 3.2 introduces the definition and operations of EVIDDs and the encoding of priority-related enabledness. The modified saturation algorithm is presented in Section 3.3, while the results of evaluation are discussed in Section 3.4. Finally, Section 3.5 provides concluding remarks and our plans for future work.

3.1.1 Overview

In this section, we investigate the problem of state space generation for models with priorities. Our goal is to efficiently build the handling of priorities into saturation – which in its original form does not consider priorities directly.

Previous work has addressed this problem by encoding the effect of priorities into the transition relations. In [Min04], the author had two main goals. Firstly, matrix diagrams have been introduced to encode the transition relations, thus relaxing the requirement of having to use Kroenecker-consistent next-state relations. This was necessary because the modification of the relations to exclude states in which a higher-priority transition is enabled almost always spoils Kroenecker-consistency. Although it is possible to decompose such a relation into Kroenecker-consistent relations, this was deemed inefficient (see Section 2.3.4 for more details about next-state representations).

Secondly, [Min04] has also pointed out that the modified next-state relations lose the property of locality. With regard to saturation, this means a drastic raise in the *Top* values of events, degrading saturation to BFS or chaining BFS. This problem has been alleviated by slicing the relations to extract the part which really depends on the additional components and keeping the rest lower. This way they have managed to preserve locality as much as possible without modifying the saturation algorithm.

On the contrary, we chose to extend saturation and use every next-state relation as if transitions were not prioritized, in the hopes of achieving better scalability. Assuming the priorities are given as integers (contrary to [Min04] but in accordance with [TFP03]), the highest priority among enabled transitions $\pi_{\max}$, or more intuitively, the minimal priority to fire is encoded into a separate data structure for every possible marking. This information is passed along with recursive calls in a modified saturation algorithm and used to decide whether a transition can be fired, similarly to the passing of constraints in constrained saturation (as described in Section 2.4.3).

The minimal priority $\pi_{\max}(M)$ to fire depends on the current marking M of the Petri net. Thus the encoding must be suitable to compute $\pi_{\max}$ for any marking M encountered by saturation, in one of the following ways.

Firstly, an overapproximation $\hat{\mathcal{S}}_r$ of the prioritized model's reachable state space $\mathcal{S}_r$ can be calculated. As saturation only encounters reachable markings $M \in \mathcal{S}_r$, it is sufficient to encode $\pi_{\max}(M)$ for the elements of $\hat{\mathcal{S}}_r$. The approximation may come from knowing bounds of places *a priori*, deriving bounds from P-invariants (see [Mur89] for details about invariants) or exploring the state space of the unprioritized version of the model. However, this calculation may not always be possible, e. g. due to lack of known place bounds or the unprioritized model being unbounded. Moreover, poor overapproximations may produce unnecessarily large encodings.

Secondly, the encoding of $\pi_{\max}(M)$ may be calculated on the fly. When saturation encounters a new local state, the data structure can be updated accordingly. We aim to explore this approach in future work.

Thirdly, a specialized data structure may be introduced that can encode $\pi_{\max}$ for any reachable or unreachable marking and compiled before saturation. To this end, we introduce edge-valued interval decision diagrams (EVIDD) to encode for each state the maximum of the priorities of enabled transitions (Section 3.2). We show that in case of Petri nets this information can be compiled offline (Section 3.2.2). The extended saturation algorithm and a more detailed comparison of our approach and that of [Min04] will be discussed in Section 3.3.

3.2 Edge-valued Interval Decision Diagrams

This section introduces edge-valued interval decision diagrams, a hybrid between edge-valued decision diagrams [RS10] and interval decision diagrams [Tov08].

Definition 27 (Edge-valued interval decision diagram) An ordered *edge-valued interval decision diagram* (EVIDD) over K variables is a tuple $\langle K, \mathcal{V}, H, r, r_h, \mathit{lvl}, \mathit{edges}\rangle$ such that:

- $\mathcal{V} = \bigcup_{k=0}^{K} \mathcal{V}_k$ is the set of *nodes* with $\mathcal{V}_0 = \{\mathbf{1}\}$ (the single *terminal node*) and $\mathcal{V}_{\geq 1} = \mathcal{V} \setminus \mathcal{V}_0$ being the set of *internal nodes*;
- $H = \bigcup_{k=0}^{K} H_k$ is the set of *handles* where $H_k = \mathbb{N} \times (\mathcal{V}_k \cup \{\mathbf{1}\})$, i. e. every handle is a pair of a *value* and a node;
- $\mathit{lvl}\colon (\mathcal{V} \cup H) \to \{0, 1, \dots, K\}$ assigns non-negative *level numbers* to each node and handle, associating them with the variables ($\mathcal{V}_k = \{n \in \mathcal{V} \mid \mathit{lvl}(n) = k\}$ and $H_k = \{h \in H \mid \mathit{lvl}(h) = k\}$);
- The root node r is the single node on level K ($\mathcal{V}_K = \{r\}$) and $r_h = \langle v, r\rangle$ is the root handle with value v, representing the encoded function;
- $\mathit{edges}\colon \mathcal{V}_{\geq 1} \to (\mathbb{N} \times H)^*$ assigns an *edge list* (a sequence of edges) to internal nodes, i. e. for any node $n \in \mathcal{V}_{\geq 1}$, $\mathit{edges}(n) = \big(\langle \mathit{lb}_1, h_1\rangle, \dots, \langle \mathit{lb}_c, h_c\rangle\big)$, c denoting the number of edges of n. Each edge consists of a *lower bound* lb_i and a handle h_i such that $h_i \in H_{k-1}$. We require that $\mathit{lb}_1 = 0$ and for all $1 < i \leq c : \mathit{lb}_{i-1} < \mathit{lb}_i$, i. e. the lower bounds form an increasing sequence. ▪

An EVIDD may be represented by a directed graph (see the bottom row of Figure 3.3 for examples). Internal nodes of the EVIDD have several outgoing edges. Each edge $\langle \mathit{lb}_i, h_i\rangle \in \mathit{edges}(n)$ is labelled with a lower bound lb_i and value v of the handle $h_i = \langle v, m\rangle$, connecting n to m. The terminal node $\mathbf{1}$ has no outgoing edge.

If $\langle v, n\rangle$ is a handle and $w \in \mathbb{N}$, let $\langle v, n\rangle + w$ and $\langle v, n\rangle - w$ denote $\langle v + w, n\rangle$, $\langle v - w, n\rangle$, respectively. The latter is defined only when $w \leq v$.

The edge lower bounds lb_i of some internal EVIDD node n partition $\mathbb{N}$ into disjoint intervals $[\mathit{lb}_1 = 0, \mathit{lb}_2), [\mathit{lb}_2, \mathit{lb}_3), \dots, [\mathit{lb}_{c-1}, \mathit{lb}_c), [\mathit{lb}_c, \infty)$. For convenience we will write $\mathit{lb}_{c+1} = \infty$. For any $x \in \mathbb{N}$ there is a unique highest index i of $\mathit{edges}(n)$ such that $\mathit{lb}_i \leq x$, which corresponds to the interval $[\mathit{lb}_i, \mathit{lb}_{i+1})$ containing x. Let $\langle v, n\rangle[x] = h_i + v$, where $\langle \mathit{lb}_i, h_i\rangle \in \mathit{edges}(n)$ and i is the index defined above. Moreover, let $\langle v, \mathbf{1}\rangle[x] = \langle v, \mathbf{1}\rangle$ for any x.

Definition 28 (Semantics of EVIDD) An EVIDD rooted in handle h encodes the function $g_h\colon \mathbb{N}^K \to \mathbb{N}$ such that $g_h(\mathbf{x}) = g_h(x_K, \dots, x_1) = w$, iff $\langle w, \mathbf{1}\rangle = h[\mathbf{x}] = h[x_K][x_{K-1}]\cdots[x_1]$, where $\mathbf{x} \in \mathbb{N}^K$. ▪

Since $h[x] \in H_{k-1}$ for all $h \in H_k$, the result of K-fold indexing is always defined for root handles and it always returns a handle of the form $\langle w, \mathbf{1}\rangle$.

Lemma 1 For every suffix $\mathbf{x}_{\geq k} = (x_k, x_{k+1}, \dots, x_K)$ of $\mathbf{x}$, $g_h(\mathbf{x}_{\geq k}) \leq g_h(\mathbf{x})$. ∎

Proof *Due to nonnegative edge values, $h[\mathbf{x}_{\geq k}] = \langle z, m\rangle$ implies $g_h(\mathbf{y}) \geq z$ for all $\mathbf{y} = (y_1, y_2, \dots, y_{k-1}, \mathbf{x}_{\geq k})$. Note that if $h[\mathbf{x}_{\geq k}] = \langle z, \mathbf{1}\rangle$, then $g_h(y) = z$.*

Definition 29 An internal EVIDD node $n \in V_{\geq 1}$ is *canonical* if *1)* for all adjacent edges $(\langle lb_i, h_i\rangle, \langle lb_{i+1}, h_{i+1}\rangle) \subseteq$ *edges*(n), $h_i \neq h_{i+1}$ and *2)* there is an edge $\langle lb_i, \langle v_i, m_i\rangle\rangle \in$ *edges*(n) such that $v_i = 0$; The terminal EVIDD node $\mathbf{1}$ is *canonical.* An (ordered) EVIDD is *quasi-reduced* if *1)* all nodes are canonical, *2)* no two internal nodes have equal edge lists and *3)* if the following holds: if *edges*$(n) = (\langle 0, \langle v_1, m_1\rangle\rangle)$ for some internal node n, then $m_1 \neq \mathbf{1}$. ∎

In the rest of this chapter we assume all EVIDDs to be quasi-reduced and ordered.

The following lemma shows that the handle h uniquely represents g_h, which means caching may be used to speed up operations with functions g_h.

Lemma 2 Let $h = \langle v, n\rangle$ and $q = \langle w, m\rangle$ be handles of nodes in a quasi-reduced ordered EVIDD such that $h, q \in H_k$. If $g_h(\mathbf{x}) = g_q(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^k$, then $h = q$. ∎

Proof *We proceed by induction by increasing k. If $k = 0$, the claim is trivial.*

In the inductive case, we need to consider handles $h \in H_k \setminus H_0$. Thanks to the induction hypothesis, it suffices to show that $h[x] = q[x]$ for all $x \in \mathbb{N}$ implies $h = q$. Let x be such that v' is minimized in $h[x] = q[x] = \langle v', n'\rangle$. Then $v' = v + \min(v_j) = w + \min(w_i)$, where v_i and w_i range over the edge values of n and m, respectively. For canonical n and m, $\min(v_i) = \min(w_i) = 0$, thus $v = w$.

Now we show that edges(n) = edges(m), which implies $n = m$. Consider some $j \in \mathbb{N}$ such that $h[j-1] \neq h[j]$. Then $\langle j, h[j] - v\rangle$ must appear in edges(m). Conversely, if $h[j-1] = h[j]$ and m is canonical, no edge with lower bound j may appear in edges(m). Finally, note that the first element of edges(m) is $\langle 0, h[0] - v\rangle$, which is also the first element on edges(m).

3.2.1 EVIDD Operations

3.2.1.1 Building Canonical EVIDDs

Fig. 3.1a shows the procedure EvIDDCheckIn that creates a canonical EVIDD node from a list of edges. Callers must ensure that the edge list contains no invalid level skipping, i. e. all child nodes are located on the same level or are the terminal node $\mathbf{1}$. Adjacent edges with equal values and child nodes are removed in lines 4–6. If only a single edge to $\mathbf{1}$ remains, a handle to the terminal node is returned instead of a new node in line 10. Otherwise, the edge list is brought into canonical form in lines 12–13 by subtracting *offset* $= \min(v_i)$ from the edge values so that a zero valued edge appears.

Lines 14–15 depend on three other routines to produce a node object in memory. The constructor EvIDDNode(E) creates a new node object from a canonical list of edges E. As in other decision diagram implementations, space is conserved and comparisons of nodes are made more efficient by the use of a unique table. The function CheckIn handles the unique table – if it contains a node with the same edges as n, CheckIn(n) disposes of the object pointed by n and returns a reference to the equivalent node from the unique table. Otherwise n is returned and CheckIn(n) adds n to the unique table. Finally, *offset* is recovered as the value of the returned handle $\langle$*offset*$, n\rangle$.

```
Input: edges $E = (\langle lb_i, \langle v_i, m_i \rangle\rangle)_{i=1}^{c}$
Output: checked in EVIDD handle
if $lb_1 \neq 0$ then fail
for $i \leftarrow 2$ to $c$ do
    if $lb_{i-1} \geq lb_i$ then fail
    if $v_i = v_{i-1}$ and $m_i = m_{i-1}$ then
        drop $\langle lb_i, \langle v_i, m_i \rangle\rangle$ from $E$
        $i \leftarrow i - 1, c \leftarrow c - 1$
    end
end
if $c = 1$ and $m_1 = \mathbf{1}$ then
    return $\langle v, \mathbf{1} \rangle$
end
$offset \leftarrow \min_{i=1,2,\dots,c} v_i$
for $i \leftarrow 1$ to $c$ do $v_i \leftarrow v_i - offset$
$n \leftarrow$ EviddNode$(E)$
$n \leftarrow$ CheckIn$(n)$
return $\langle offset, n \rangle$
```

(a) Procedure EviddCheckIn.

```
Input: $a = \langle v, n \rangle, b = \langle w, m \rangle \in H_k$
Output: $\max\{a, b\}$
if $n = \mathbf{1}$ and $m = \mathbf{1}$ then
    return $\langle \max\{v, w\}, \mathbf{1} \rangle$
end
$offset \leftarrow \min\{v, w\}$
$a \leftarrow a - offset, b \leftarrow b - offset$
if $\neg$MaxCacheGet$(\{a, b\}, h)$ then
    if $n = \mathbf{1}$ then
        $h \leftarrow$ MergeConstant$(b, v)$
    else if $m = \mathbf{1}$ then
        $h \leftarrow$ MergeConstant$(a, w)$
    else
        $h \leftarrow$ Merge$(a, b)$
    end
    MaxCachePut$(\{a, b\}, h)$
end
return $h + offset$
```

(b) Procedure Maximum.

Figure 3.1: Basic EVIDD operations.

```
Input: $a = \langle v, n \rangle$ and $w \in \mathbb{N}$
Output: $\max\{a, \langle w, \mathbf{1} \rangle\}$
$E \leftarrow (\,)$
for each $\langle lb_i, h_i \rangle \in edges(n)$ do $E \leftarrow E \mathbin{+\!\!+} (\langle lb_i,$ Maximum$(h_i + v, \langle w, \mathbf{1} \rangle)\rangle)$
return EviddCheckIn$(E)$
```

(a) Procedure MergeConstant.

```
Input: $a = \langle v, n \rangle, b = \langle w, m \rangle \in H_k$
Output: $\max\{a, b\}$
$c \leftarrow |edges(n)|, c' \leftarrow |edges(m)|, i \leftarrow 1, j \leftarrow 1, E \leftarrow (\,), lb_{\text{out}} \leftarrow 0$
let us denote $edges(n)$ by $(\langle lb_k, h_k \rangle)_{k=1}^{c}$ and $edges(n)$ by $(\langle lb'_k, h'_k \rangle)_{k=1}^{c'}$
while $i \leq c$ and $j \leq c'$ do
    $E \leftarrow E \mathbin{+\!\!+} (\langle lb_{\text{out}},$ Maximum$(h_i + v, h'_j + w)\rangle)$
    if $i = c$ then $nextA \leftarrow \infty$ else $nextA \leftarrow lb_{i+1}$
    if $j = c'$ then $nextB \leftarrow \infty$ else $nextB \leftarrow lb'_{j+1}$
    $lb_{\text{out}} \leftarrow \max\{nextA, nextB\}$
    if $nextA = lb_{\text{out}}$ then $i \leftarrow i + 1$
    if $nextB = lb_{\text{out}}$ then $j \leftarrow j + 1$
end
return EviddCheckIn$(E)$
```

(b) Procedure Merge.

Figure 3.2: Subroutines for the Maximum operation ($\mathbin{+\!\!+}$ denotes concatenation).

3.2.1.2 Elementwise Maximum

Definition 30 The *elementwise maximum* of the EVIDD handles $a, b \in H_k$ is the handle $h = \max\{a, b\}$, such that $\max\{g_a(\mathbf{x}), g_b(\mathbf{x})\} = g_h(\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^k$. ∎

The semantics of EVIDDs together with the definition of $\max\{a, b\}$ imply that $\max\{g_a(\mathbf{x}), g_b(\mathbf{x})\} = \max\{g_{a[x_k]}(\mathbf{x}_{\leq k-1}), g_{b[x_k]}(\mathbf{x}_{\leq k-1})\}$. Therefore $\max\{a, b\}[x] = \max\{a[x], b[x]\}$ for all $x \in \mathbb{N}$, which allows recursive calculation of $\max\{a, b\}$. The operation has two further properties which will be exploited in our implementation to facilitate caching. Firstly, the operation is symmetric: $\max\{a, b\} = \max\{b, a\}$. Secondly, because $q = h + w$ implies $g_q(\mathbf{x}) = g_h(\mathbf{x}) + w$ for all $\mathbf{x}$, the elementwise maximum is *offset invariant*. If $h = \max\{a, b\}$, we have $h + w = \max\{a + w, b + w\}$ and $h - w = \max\{a - w, b - w\}$.

Fig. 3.1b shows the implementation Maximum of the elementwise maximum operation. The algorithm is divided into four cases based on whether the handles a and b point to terminal or internal EVIDD nodes. If a and b are both handles of the terminal node $\mathbf{1}$ (line 1), the functions g_a and g_b are constant. This base case is processed directly without caching. The remaining recursive cases make use of caching. Maximum depends on the routines MaxCacheGet and MaxCachePut to manage the cache. MaxCacheGet$(\{a, b\}, h)$ takes an unordered caching key $\{a, b\}$ and sets the reference h to the cached result $\max\{a, b\}$. Successful retrievals are indicated by returning **true**, while **false** is returned on cache misses. MaxCacheGet$(\{a, b\}, h)$ associates the result h with the key $\{a, b\}$.

To increase the number of potential cache hits, lines 4–5 subtract the minimum of their values from the handles $a = \langle v, n\rangle$ and $b = \langle w, m\rangle$, so that at least one of v and w is 0. After possibly retrieving $\max\{a, b\}$ from the cache, this *offset* is added back to the result in line 16.

The function MergeConstant in Fig. 3.2a processes the two cases when one of a and b is a handle to $\mathbf{1}$, while the the other references an internal node. Due to symmetry, we may assume that $a = \langle v, n\rangle \in H_k$ and $b = \langle w, \mathbf{1}\rangle \in H_k$. Because $\langle w, \mathbf{1}\rangle[x] = \langle w, \mathbf{1}\rangle$, $\max\{a, b\}[x]$ must be set to $\min\{a[x], \langle w, \mathbf{1}\rangle\}$ for all $x \in \mathbb{N}$. This is accomplished by replacing all edges $\langle lb_i, h_i\rangle$ of n with $\max\{a[lb_i], \langle w, \mathbf{1}\rangle\}$.

The most interesting case, when the handles $a = \langle v, n\rangle$, $b = \langle w, m\rangle$ both refer to internal nodes $n, m \in V_k$ is processed by Merge in Fig. 3.2b. The difficulty arises from the edge lists $edges(n) = (\langle lb_k, h_k\rangle)_{k=1}^{c}$ and $edges(m) = (\langle lb'_k, h'_k\rangle)_{k=1}^{c'}$ having possibly different lower bound sequences lb_i and lb'_j. Therefore a new edge list E with a new sequence of lower bounds $\{lb_i\} \cup \{lb'_j\}$ must be constructed.

Since $lb_1 = lb'_1 = 0$, the first edge of the new edge list is $\langle 0, \max\{a[0], b[0]\}\rangle = \langle 0, \max\{h_1 + v, h'_1 + w\}\rangle$. The loop in lines 3–9 of Merge traverses the lower bounds lb_i and lb'_j with the indices i and j. Lines 5 and 6 peek at the next elements $nextA = lb_{i+1}$ and $nextB = lb'_{j+1}$ of the lower bound sequences. We follow the convention that $lb_{c+1} = lb'_{c'+1} = \infty$. The lower bound lb_{out} of the next edge to be created is equal to the smaller of the two next elements. Thus an intersection of the interval partitions of $\mathbb{N}$ induced by $edges(n)$ and $edges(m)$ is built. If both edge lists are exhausted, $lb_{\text{out}} = nextA = nextB = \infty$, which causes both i and j to be incremented beyond their limits and the loop to terminate.

Other elementwise operations for EVIDD handles can be implemented similarly. Elementwise minimum is straightforward, since it is both symmetric and offset invariant. However, for operations which are offset dependent, such as addition, or asymmetric, such as subtraction, the caching logic must be adjusted accordingly.

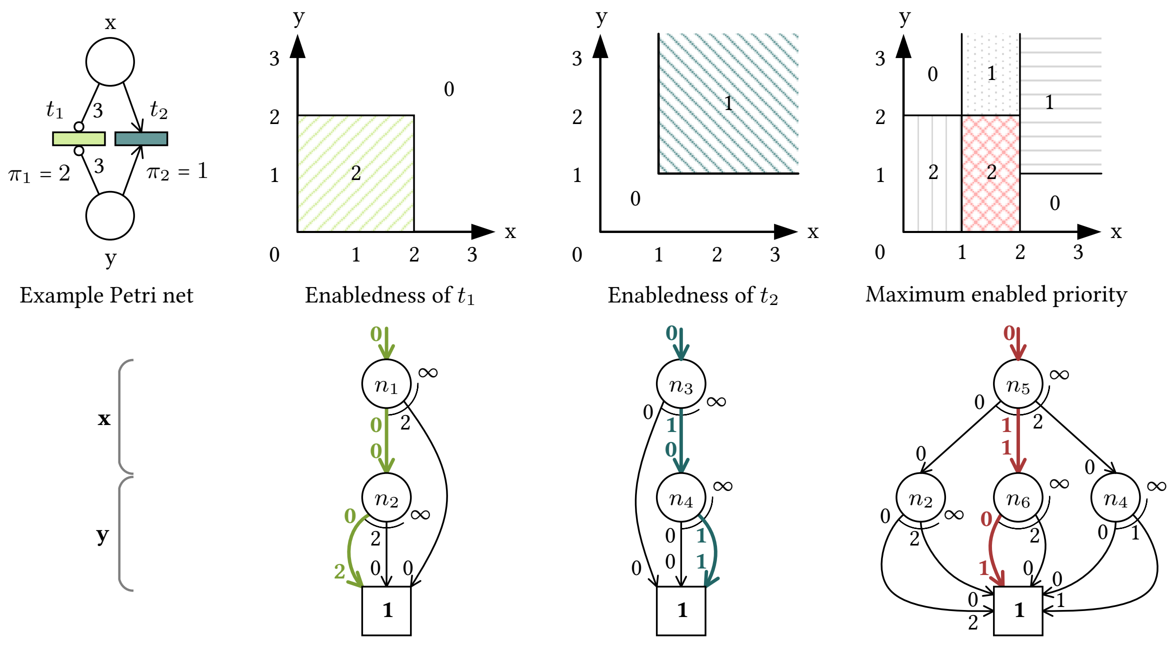


Figure 3.3: Regions of the space of markings defined by (transitions of) the example Petri net with priorities (top row) and corresponding EVIDD representations (bottom row). Highlighted regions are encoded by the highlighted paths in the EVIDD.

3.2.2 Encoding the Minimal Priority to Fire

In this section we construct an EVIDD and a handle h that encodes the minimal priority to fire a transition of a prioritized Petri net for any state. We will have $g_h(M(p_K), M(p_{K-1}), \dots, M(p_1)) = \pi_{\max}(M)$ for a marking M of the Petri net if a transition with priority $\pi_{\max}$ has the highest priority among all enabled transitions in M. If there are no enabled transitions in M, we set $\pi_{\max}(M) = 0$.

TransitionHandle(t), which is shown in Fig. 3.4a, associates an EVIDD handle h to a prioritized Petri net transition t. The handle encodes the function

$$g_h(M(p_K), M(p_{K-1}), \dots, M(p_1)) = \begin{cases} \pi(t), & \text{if } M \in \mathit{En}(t), \\ 0, & \text{if } M \notin \mathit{En}(t), \end{cases}$$

where $\mathit{En}(t)$ is the set of markings in which t is enabled:

$$\mathit{En}(t) = \prod_{k=1}^{K} \left[W^-(t, p_k), W^\circ(t, p_k)\right) \cap \mathbb{N}^K,$$

i. e. $\mathit{En}(t)$ is the set of integer vectors where the component corresponding to the place p_k lies in the interval $\left[W^-(t, p_k), W^\circ(t, p_k)\right)$. Recall that if there are no inhibitor edges between t and the place p_k, then $W^\circ(t, p_k) = \infty$. If $\pi(t) = 0$ or $\mathit{En}(t) = \varnothing$, g_h is constant and h is $\langle 0, \mathbf{1}\rangle$.

These intervals are encoded by the loop in lines 3–8 from the lowest to the top level of the EVIDD. If t is never enabled due to an empty interval, a zero handle is returned in line 4. The function checks in handles $h^{(k)} \in H_k$ such that $g_{h^{(k)}}(\mathbf{x}_{\le k}) = \pi(t)$ for all $\mathbf{x}_{\le k} \in \prod_{i=1}^{k}\left[W^-(t, p_i), W^\circ(t, p_i)\right) \cap \mathbb{N}^i$, otherwise 0. For all $k < \mathit{Bot}(t)$, $h^{(k)} = \langle \pi(t), \mathbf{1}\rangle$ due to the reduction of zero nodes in EviddCheckIn. Moreover, for all $k > \mathit{Top}(t)$, $h^{(k)} = \langle \pi(t), n\rangle$, where $\mathit{edges}(n) = (\langle 0, h^{(k-1)}\rangle)$ and the EVIDD is a single path, because $W^-(t, p_k) = 0$ and $W^\circ(t, p_k) = \infty$.

```
Input: transition $t$
Output: priority EVIDD handle
if $\pi(t) = 0$ then return $\langle 0, \mathbf{1} \rangle$
$h^{(0)} \leftarrow \langle \pi(t), \mathbf{1} \rangle$
for $k \leftarrow 1$ to $K$ do
    if $W^{-}(t, p_k) > W^{\circ}(t, p_k)$ then return $\langle 0, \mathbf{1} \rangle$
    if $W^{-}(t, p_k) > 0$ then $E \leftarrow (\langle 0, \langle 0, \mathbf{1} \rangle \rangle, \langle W^{-}(t, p_k), h^{(k-1)} \rangle)$
    else $E \leftarrow (\langle 0, \langle 0, h^{(k-1)} \rangle \rangle)$
    if $W^{\circ}(t, p_k) < \infty$ then $E \leftarrow E \mathbin{+\!\!+} (\langle W^{\circ}(t, p_k), \langle 0, \mathbf{1} \rangle \rangle)$
    $h^{(k)} \leftarrow$ EviddCheckIn($E$)
end
return $h^{(K)}$
```

(a) Procedure TransitionHandle.

```
Input: set of all transitions $T$
Output: EVIDD handle encoding the minimal priority to fire
$h \leftarrow \langle 0, \mathbf{1} \rangle$
order $T$ by $Top(t)$ nondecreasing
for each $t \in T$ do
    $q \leftarrow$ TransitionHandle($t$)
    $h \leftarrow$ Maximum($h, q$)
end
return $h$
```

(b) Procedure RequiredPriority.

Figure 3.4: Encoding the highest priority of enabled transitions.

RequiredPriority in Fig. 3.4b encodes $\pi_{\max}$ as an EVIDD handle. For each transition t the EVIDD handle describing the enabling states $En(t)$ and the priority $\pi(t)$ is constructed by TransitionHandle. The Maximum operation is used to merge the transition handles into a single handle. Analogously to a heuristic in constraint programming with MDDs [c15], Maximum is called for the transition handles ordered by $Top(t)$ nondecreasing. Hence upper levels of the EVIDD are left as a single path for as long as possible, which we have found to improve performance.

Figure 3.3 illustrates the above concepts on a simple example. The enabling region of a transition is an (infinite) box in the K dimensional space of Petri net markings, labeled with its priority, where K is the number of places in the net. For the sake of clarity, we use a simple Petri net with two places. Therefore regions are rectangles or quadrants in 2-dimensional space, as seen in the top row. With one or more sets of potentially overlapping regions corresponding to every transition in the net (t_1 and t_2 in this case), the Maximum operation computes a new set of disjoint regions that cover exactly the same points as the original regions (rightmost part of Figure 3.3), where labels are obtained as the maximum of labels on original regions that cover the new region. For regions that do not enable any transition we assign the priority level 0.

The bottom row of Figure 3.3 shows the EVIDD representations of the sets of regions above them. According to Definition 27, a *handle* (at the arrowhead of an arc) is a pair of a *node* and a *weight*. The weight represents a portion of the priority of the highest enabled transition belonging to the region such that the priority is the sum of weights along the path. A node represents a dimension (in this case a marking of a place) and has an ordered list of *arcs* partitioning the dimension: each arc corresponds to an interval with an inclusive lower bound defined by its label and exclusive upper bound defined by the label of the next arc of the node, or infinity (which is not associated to an arc), and points to a *child handle* for the next dimension or the terminal node. Weights are distributed automatically by reduction rules (see Definition 29).

3.3 Saturation with Priority Constraints

In this section, we discuss our extension in detail and also give some remarks about its advantages over previous approaches. Because we can now represent the priority related enabledness of the transitions, we will use the simplest representation for unprioritized next-state relations. As mentioned in Section 2.3.4, this will be an implicit relation forest with three integers characterizing the partial functions.

Definition 31 (Implicit relation forest for Petri net transitions) An implicit relation forest (IRF) for Petri net transitions is a tuple $\mathit{IRF} = (\mathcal{V}, \mathit{lvl}, W^-, W^\circ, W^+, \mathit{next})$, where:

- $\mathcal{V} = \bigcup_{k=0}^{K} \mathcal{V}_k$ is the set of *nodes*, where items of $\mathcal{V}_0$ are the *terminal* nodes **1** and **0**, the rest ($\mathcal{V}_{>0} = \mathcal{V} \smallsetminus \mathcal{V}_0$) are *internal* nodes ($\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ if $i \neq j$);
- $\mathit{lvl}\colon \mathcal{V} \to \{0, 1, \dots, K\}$ assigns non-negative *level numbers* to each node, associating them with variables according to the variable ordering (nodes in $\mathcal{V}_k = \{r \in \mathcal{V} \mid \mathit{lvl}(r) = k\}$ belong to variable x_k for $1 \leq k \leq K$ and are terminal nodes for $k = 0$);
- W^-, W° and W^+ are functions from $\mathcal{V}$ to $\mathbb{N}$ and encode the effect of a transition on a place belonging to the level of the node;
- $\mathit{next} : \mathcal{V} \to \mathcal{V}$ is the next node if the implicit function is interpreted on an input, i. e. indexing a node yields $r[i, j] = \mathit{next}(r)$ if $W^-(r) \leq i < W^\circ(r)$ (the transition is enabled) and $j = i - W^-(r) + W^+(r)$ (the marking changed from i to j), while $r[i, j] = \mathbf{0}$ otherwise;
- for every pair of nodes on the same level $r_1, r_2 \in \mathcal{V}_k$, if $W^-(r_1) = W^-(r_2)$, $W^\circ(r_1) = W^\circ(r_2)$ and $W^+(r_1) = W^+(r_2)$, then $r_1 = r_2$.

Semantics are defined based on node indexing in the same way as for MxDs. For a Petri net transition t, an IRF chain assuming a variable ordering can be built from bottom to top as follows. Let $r_0 = \mathbf{0}$. We define $r_k \in \mathcal{V}_k$ such that $\mathit{next}(r_k) = r_{k-1}$, $W^-(r_k) = W^-(p_k, t)$, $W^\circ(r_k) = W^\circ(p_k, t)$ and $W^+(r_k) = W^+(p_k, t)$. ▪

This representation is implicit, therefore it does not have to be updated when a new state is found. Furthermore, it is as compact as the original Petri net. Such a Kroenecker-consistent representation would not be possible without separating the priority-related aspects from the description of the transition itself.

3.3.1 Details of the Algorithm

Given the EVIDD notation and the operations defined so far, as well as the implicit relation forest for Petri nets, Fig. 3.5 presents the pseudocode of the extended saturation algorithm capable of handling prioritized models natively. The pseudocode uses $\mathcal{E}_k^\pi = \{e \mid e \in \mathcal{E}_k \wedge \pi(e) = \pi\}$ to denote the set of events “belonging” to level k (as defined in Definition 21) and having priority π, as well as the self-explanatory $\mathcal{E}_k^{\pi \geq v}$ (v is a given priority level). The IRF node corresponding to event e and therefore encoding $\mathcal{N}_e$ without priority considerations is denoted by $r(e)$.

The procedure Saturate (Fig. 2.7a) takes an MDD node n and an EVIDD handle h – which are initially the root of the MDD representing the set of initial states ($\mathcal{I}$ in Definition 9) and the root handle of the EVIDD as returned by RequiredPriority (Fig. 3.4b) – and saturates n. Recall that when

```
Input: MDD node $n$, EVIDD handle $h = \langle v, m \rangle$
Output: saturated MDD node $n'$
if $n = \mathbf{0}$ or $n = \mathbf{1}$ then return $n$
if $\neg$SatCacheGet$(n, h, n')$ then
    $n' \leftarrow$ MddNode$(\mathit{lvl}(n))$
    for each $x$ where $n[x] \neq \mathbf{0}$ do $n'[x] \leftarrow$ Saturate$(n[x], h[x])$
    $n' \leftarrow$ CheckIn$(n')$
    repeat
        changed $\leftarrow$ false
        for each $e \in \mathcal{E}^{\pi \geq v}_{\mathit{lvl}(n)}$ do
            $n'' \leftarrow$ SatFire$(n, e, h)$
            if $n' \neq n''$ then $n' \leftarrow n''$, changed $\leftarrow$ true
        end
    until $\neg$changed
    SatCachePut$(n, h, n')$
end
return $n'$
```

(a) Procedure Saturate.

```
Input: MDD node $n$, event $e$, EVIDD handle $h = \langle v, m \rangle$
Output: the result of firing $e$ from the states $n$ with the children saturated
$\pi \leftarrow \pi(e)$, $r \leftarrow r(e)$
if $n = \mathbf{0}$ or $\pi < v$ then return $\mathbf{0}$
if $r = \mathbf{1}$ and $m = \mathbf{1}$ then
    if $\pi = v$ then return $n$ else fail "invalid descriptor"
end
$n' \leftarrow$ MddNode$(\mathit{lvl}(n))$
for each $x, y$ where $r[x, y] \neq \mathbf{0}$ do
    $s \leftarrow$ RelProdSat$(\pi, r[x, y], n[x], h[x], h[y])$
    $n'[y] \leftarrow$ Union$(n'[y], s)$
end
$n' \leftarrow$ CheckIn$(n')$
return $n'$
```

(b) Procedure SatFire.

```
Input: priority $\pi$, IRF node $r$, MDD node $n$, EVIDD handles $h = \langle v, m \rangle$, $h'$
Output: saturated MDD node $n''$, which is the result of firing $d$ from $n$
if $n = \mathbf{0}$ or $\pi < v$ then return $\mathbf{0}$
if $r = \mathbf{1}$ and $m = \mathbf{1}$ then
    if $\pi = v$ then return $n$ else fail "invalid descriptor"
end
if $\neg$RelProdCacheGet$(\pi, r, n, h, h', n'')$ then
    $n' \leftarrow$ MddNode$(\mathit{lvl}(n))$
    for each $x, y$ where $r[x, y] \neq \mathbf{0}$ do
        $s \leftarrow$ RelProdSat$(\pi, r[x, y], n[x], h[x], h'[y])$
        $n'[y] \leftarrow$ Union$(n'[y], s)$
    end
    $n' \leftarrow$ CheckIn$(n')$, $n'' \leftarrow$ Saturate$(n', h')$, RelProdCachePut$(\pi, r, n, h, h', n'')$
end
return $n''$
```

(c) Procedure RelProdSat.

Figure 3.5: Saturation with EVIDDs for prioritized models.

the root node gets saturated, it represents the set of reachable states $\mathcal{S}_r = \mathcal{N}^*(\mathcal{I})$. The procedure first recursively saturates every child node (lines 3–4). The constructor MddNode creates a new node on the current level which will hold the new (saturated) children. Similarly to EviddCheckIn, CheckIn in line 5 ensures that the resulting node n' is unique (i. e. the MDD currently being processed is quasi-reduced). Lines 6–10 perform the fixed point computation with the next-state functions corresponding to $\mathcal{E}^{\pi \geq v}_{lvl(n)}$, i. e. those events that "belong" to the current level and have a priority of at least v, the value of handle h. Note that v is indeed a lower bound of the priority of any fireable transition, as shown by Lemma 1. Terminal nodes are returned immediately.

The procedure SatFire computes the image of $\mathcal{N}_e$ on $S(n)$. RelProdSat is used to compute the image recursively for every component, also saturating new nodes during the process (line 11 of Fig. 2.7c). Due to this, both procedures return a saturated (and also quasi-reduced) node. SatFire uses the priority and the descriptor belonging to event e to evaluate base cases. If $S(n)$ is empty or the value of the priority handle h is higher than $\pi(t)$ (i. e. there is at least one enabled transition with a higher priority), the terminal zero node is returned immediately. On the other hand, if the descriptor r is the identity descriptor and the node of the handle is the terminal EVIDD node, we expect that the priority of the current transition will be v and then we can return n as is (because of the identity relation). If v is lower than the current priority, then either the descriptor or the priority EVIDD is invalid, since the event e is enabled and has higher priority than any enabled transition (including itself), which is an obvious contradiction. Lines 6–9 recursively compute the image of $\mathcal{N}_e$. RelProdSat does essentially the same, but it also saturates the resulting node before returning it (line 11 of Fig. 2.7c). Note, however, that in RelProdSat we consider two EVIDD handles – one for the source state (h) and one for the target state (h'). The former is used to evaluate the enabledness of the transition currently being fired, while the latter will be used to saturate the resulting node.

To exploit the structure of decision diagrams (i. e. the same node may be reached on multiple paths), Saturate and RelProdSat use caches to store previously computed results (lines 2, 13 of Fig. 2.7a and lines 5, 11 of Fig. 2.7c).

3.3.2 Discussion

The correctness of the presented algorithm can be proved along the following (schematic) considerations. Suppose that we decompose the next-state relation into $\mathcal{N}_e = \widehat{\mathcal{N}_e} \setminus E_e$ such that $\widehat{\mathcal{N}_e}$ is the next-state relation without considering priorities (which is by definition a superset of $\mathcal{N}_e$) and $E_e = En^{\pi > \pi(e)} \times \mathcal{S}$ where $En^{\pi > \pi(e)} = \bigcup_{e' \in \mathcal{E}^{\pi > \pi(e)}} En(e')$, i. e. the Cartesian product of the states in which an event with higher priority is enabled and the state space. The root IRF node of e encodes $\widehat{\mathcal{N}_e}$. To encode $En^{\pi > \pi(e)}$, we use the EVIDD built by RequiredPriority: by selecting only the paths to which the EVIDD assigns a value larger than $\pi(e)$, we can exactly compute $En^{\pi > \pi(e)}$.

It is easy to see that the modified saturation algorithm performs the selection whenever π is compared to the value of a handle and also computes $\mathcal{N}_e = \widehat{\mathcal{N}_e} \setminus E_e$ on the fly. Edge-labeling therefore enables the compact representation of a series of sets $En^{\pi > i}$, where every set is the superset of the previous one. Handling of intervals instead of values, on the other hand, enables us to encode the highest priority offline in case of Petri nets.

Compared to the matrix diagram-based solution of [Min04], we expect to build more compact decision diagrams in the intermediate steps. This assumption is based on the intuition that the efficiency of saturation comes from the ability to saturate nodes as low as possible, minimizing the size of the diagram before moving to the next level. Although the firing of an event is similar in the two approaches both in terms of computing the image and caching (where [Min04] has more matrix decision diagram nodes we have more EVIDD nodes to spoil the cache), the significant difference comes

```
Input: MDD node $n$, EVIDD handle $h = \langle v, m \rangle$
Output: MDD node $n'$ corresponding to tangible markings in $n$
if $v \geq 1$ then return $\mathbf{0}$
if $n = \mathbf{0}$ or $m = \mathbf{1}$ then return $n$
if $\neg$GetTangibleCacheGet$(n, h, n')$ then
    $n' \leftarrow$ MddNode$()$
    for each local state $x$ do $n'[x] \leftarrow$ GetTangible$(n[x], h[x])$
    CheckIn$(n')$, GetTangibleCachePut$(n, h, n')$
end
return $n'$
```

Figure 3.6: Procedure GetTangible.

from the iteration order of the whole saturation algorithm. Because our approach keeps the events as is (as opposed to modifying them and raising their *Top* values), it can process more transitions when saturating a node, potentially yielding a smaller (denser) diagram after every Saturate call. The confirmation of this hypothesis would require a thorough analysis of the algorithms or the observation of how the state space MDD evolves in each case. At this stage of the work, we can provide empirical measurements that seem to confirm our expectations. This idea is taken one step further by Thesis 3 in Chapter 6.

3.3.3 Application: Stochastic Petri Nets

Tangible state space generation of Generalized Stochastic Petri nets can be performed efficiently by the proposed saturation method. First, the EVIDD encoding the highest priority of enabled transitions $\pi_{\max}$ is constructed by RequiredPriority (Fig. 3.4b). The EVIDD will encode a nonzero value for each vanishing marking. Then Saturate (Fig. 2.7a) is called on the MDD with the initial marking to explore the reachable state of the GSPN. Finally, tangible states are extracted into a new MDD by simultaneously traversing the saturated MDD and the EVIDD. This approach is similar to the "elimination after generation" in [Min04].

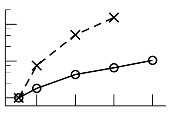
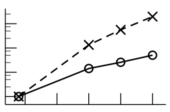
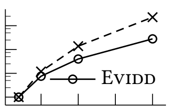
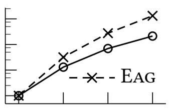
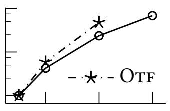
GetTangible in Fig. 3.6 extracts the tangible states $\mathcal{T}$ from the MDD corresponding to the reachable states $\mathcal{S}_r$. The function recursively traverses the MDD and the EVIDD in lines 3–6 using caches to store previously computed results. If the lower bound of $\pi_{\max}$ becomes nonzero (see Lemma 1), the zero node is returned in line 1. The recursion may also terminate in line 2 for two reasons: Firstly, the MDD **0** node may be reached. Secondly, the terminal EVIDD node **1** may be reached. This indicates that $\pi_{\max} = v = 0$ for every child of the current MDD node, therefore no further transformation is needed.

3.4 Evaluation

A prototype implementation[1] of our algorithm has been written in the Scala programming language. Measurements were run on a 2.50 GHz Intel® Xeon® L5420 processor and 32 GB memory under Ubuntu Linux 14.04. Heap space for the Java 1.8 virtual machine was maximized in 25 GB. Concurrent mark-and-sweep garbage collection was enabled in the JVM. However, no additional garbage collection routines were implemented to reclaim unique table and cache entries during saturation, i. e. MDD node collection was Lazy [CLS01].

[1]See https://inf.mit.bme.hu/en/pn2017 for more details about the measurements.

Table 3.1: Comparison with matrix diagram based methods.

			DD nodes			Comparison		
	N	$\lvert\mathcal{T}\rvert$	Final	Peak	Time	Alg.	Time	Scaling
Phils	16	4.87×10^{6}	1188	10 662	0.216 s		1.3 s	
	30	3.46×10^{12}	2364	26 086	0.390 s	Eag	10.1 s	
	60	1.20×10^{25}	4884	75 449	0.930 s		69.2 s	
	90	4.15×10^{37}	7404	147 772	1.420 s	[Min06]	204.4 s	
	120	1.44×10^{50}	9924	238 976	2.261 s		—	
Kanban	8	4.23×10^{7}	280	1800	0.045 s		0.5 s	
	30	2.36×10^{12}	1985	21 259	0.638 s	Eag	67.0 s	
	40	2.86×10^{13}	3240	41 464	1.151 s	[Min06]	280.0 s	
	50	2.01×10^{14}	4795	71 569	2.252 s		979.0 s	
FMS	8	4.46×10^{7}	280	5972	0.186 s		0.2 s	
	20	8.83×10^{9}	3646	45 031	1.407 s	Eag	2.5 s	
	40	4.97×10^{12}	13 276	232 061	7.413 s	[Min06]	29.0 s	
	80	3.71×10^{15}	50 536	1 352 121	52.009 s		477.0 s	
Poll	5	5.91×10^{6}	279	2806	0.056 s		0.4 s	
	10	9.34×10^{16}	1604	30 602	0.726 s	Eag	13.0 s	
	15	2.28×10^{28}	4729	135 267	3.867 s	[Min06]	113.1 s	
	20	3.20×10^{40}	10 404	398 512	11.831 s		540.1 s	
Courier	10	4.25×10^{9}	1433	17 703	0.626 s		14 s	
	20	2.26×10^{12}	4193	55 458	2.666 s	Otf	82 s	
	40	2.18×10^{15}	13 913	191 268	14.789 s	[Min04]	668 s	
	60	1.44×10^{17}	29 233	407 478	42.847 s		—	

3.4.1 Benchmark Models

We used several scalable families of GSPN models from the literature as benchmarks. As only the state space of the models are explored, transition timings were ignored and only transition priorities were kept. *Phils* is the modified version of the dining philosophers model from [Min04], where the action of picking up a fork is an immediate transition. The prioritized versions of the *Kanban*, *FMS* and *Poll* models were also taken from [Min04]. In particular, the *FMS* model was modified from its original version in [CT93] by setting marking-dependent arc weights to constant. *Courier* describes Courier protocol software from [WL91]. We follow [Min06] by setting $N = M$.

Phils is grown structurally, i. e. by repeating submodels, for increasing values of N. *Poll* is grown both structurally and by increasing initial token counts, while the rest of the model families grow only by initial marking.

No further modifications were needed to analyze the models. We choose as variables the marking of single places such that the highest priority of enabled transitions can be encoded as an EVIDD.

Table 3.2: Unique table and cache utilization for the *Courier* model.

		EVIDD			MDD				Otf [Min04]	
	N	$\pi_{\max}$	Peak	Cache	$\mathcal{S}_r$	$\mathcal{T}$	Peak	Cache	Peak	Cache
Courier	10	69	538	424	3236	1433	17 165	85 414	71 735	304 612
	20	69	538	424	9346	4193	54 920	264 639	227 230	857 572
	40	69	538	424	30 566	13 913	190 730	891 589	801 920	2 656 692
	60	69	538	424	63 786	29 233	406 940	1 876 539	–	–

3.4.2 Comparison with Matrix Diagram Methods

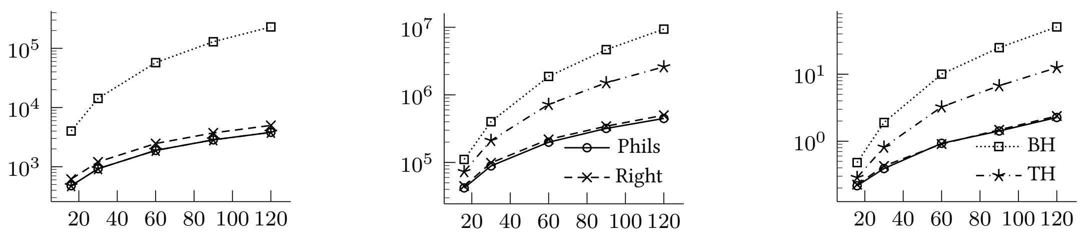
Table 3.1 shows the number of decision diagram nodes and the running times of our algorithm when applied to generate the tangible state space $\mathcal{T}$ as described in paragraph *Application: Stochastic Petri Nets* of Section 3.3. Unfortunately, we were unable to directly compare our algorithm to matrix diagram based approaches [Min04; Min06] implemented in SMART [Cia+06], as the currently available version of SMART does not support prioritized models. We instead compare to the results published in [Min04] and [Min06]. For *Courier*, we compare with the best-scaling approach from [Min04], Otf. For the other models, we compare with "elimination after generation" (Eag) from [Min06]. To account for differences between the hardware used, the semi-log plots in the Scaling column show normalized running times. The running times for each algorithm and model family were divided by the running time of the algorithm on the smallest model of the family before plotting. For example, the running time of Eag on *Phils* was divided by 1.3 s, while the running time of our algorithm was divided by 0.216 s.

Our measurements indicate that our EVIDD-based modified saturation approach scales better than matrix diagram based approaches that handle priorities by changing the next-state relations. Scaling is especially good with the structurally grown *Phils* family. To obtain a more accurate comparison, further measurements would be needed.

Table 3.2 shows the number of decision diagram nodes required for representing the highest priority of enabled transitions $\pi_{\max}$, the reachable states $\mathcal{S}_r$ and the tangible states $\mathcal{T}$, as well as the unique table and cache utilizations on the *Courier* model family. When comparing with the utilizations of Otf published in [Min04], it is apparent that – in accordance with our expectations – prioritized saturation with EVIDDs requires the creation of less temporary MDD nodes and therefore reduces the size of the cache as well (even though using pairs as keys would obviously lead to worse cache coherence in itself).

3.4.3 Models with Many Priority Levels

To study the effects of more complicated priority structures, we created three additional modifications of the *Phils* model family where we assign multiple priority levels to transitions. In these models, the picking up of a fork is an immediate event with $\pi \geq 0$, while the rest of the behaviours are timed with $\pi = 0$. In *PhilsRight*, picking up the left fork has priority 1, while picking the right fork has priority 2. In *PhilsBH* and *PhilsTH*, picking up the two forks have equal priorities. However, in *PhilsBH*, philosophers have sequentially increasing priority from the top to the bottom of the EVIDD and MDD variable order. In *PhilsTH* the order is reversed. All models have the same tangible states. Moreover, *PhilsBH* and *PhilsTH* have isomorphic reachable state spaces, albeit with different variable ordering.



(a) $\pi_{\max}$ EVIDD nodes (b) Total cache size (c) Execution time (s)

Figure 3.7: Measurements with many priority levels.

Fig. 3.7 shows the number of EVIDD nodes required to encode $\pi_{\max}$, the total number of cache entries created, and the execution time of the tangible state space generation. Adding another priority level in *PhilsRight* increased only the number of EVIDD nodes by a constant factor. The effects of assigning sequential priorities to philosophers heavily depended on the order of priorities. EVIDDs could encode priorities increasing from bottom to top in *PhilsTH* with the same number of nodes as *Phils*; however, the reversed order in *PhilsBH* increased node count substantially.

While *PhilsRight* only increased cache usage moderately compared to *Phils*, the more complicated effective next-state relations of *PhilsTH* and *PhilsBH* required much more cache entries in saturation. This problem is further amplified by the large number of EVIDD nodes that appear in cache keys in *PhilsBH*. This effect also manifests in the running times, which were found to be strongly correlated ($R = 0.999$) with the number of cache entries.

3.5 Summary and Future Work

In this thesis I have introduced a modified saturation algorithm capable of natively handling prioritized models. To this end, the chapter introduced edge-valued interval decision diagrams which can efficiently encode the priority-related enabledness of transitions and can be constructed before state space generation in case of Petri nets. I have described the new algorithm in detail and compared the results of our empirical experiments to the results of [Min04], demonstrating that handling priorities separately can indeed yield smaller intermediate diagrams and better performance.

Thesis 1 I designed an algorithm to help the efficient analysis of Generalized Stochastic Petri Nets (GSPN). It extends the traditional saturation algorithm to perform a more efficient symbolic state space exploration of systems with prioritized transitions.

1.1 I introduced a new type of decision diagram called Edge-Valued Interval Decision Diagram (EVIDD) that can encode enabledness of prioritized transitions in GSPNs.

1.2 I extended the saturation algorithm to handle prioritized transitions efficiently using an EVIDD instead of encoding priorities in the transition relation.

1.3 I evaluated the algorithm and showed that it scales better than previously known approaches.

As the direct follow-up of this work, we plan to define a full workflow to efficiently analyze the stochastic behavior of large GSPNs, also supporting phase-type distributions and marking-based behavior (e. g. in the form of marking-dependent arc weights).

Chapter 4

Saturation-Based On-the-Fly Computation of Synchronous Products for LTL Model Checking

4.1 Introduction

Linear temporal logic (LTL) specifications (presented in Section 2.1.4.2) play an important role in the history of verification as being a prevalent formalism to specify the requirements of reactive and safety-critical systems [Pnu77]. Section 2.2.5 discussed how the behaviors defined by an LTL property can be expressed with the help of *Büchi automaton* and that the problem of checking LTL properties is usually reduced to deciding language emptiness of the synchronous product of two Büchi automata: one characterizing the possible behaviors of the system and another accepting behaviors that violate the desired property [VW86]. The language emptiness problem of the result product Büchi automaton can be decided by finding *strongly connected components* (SCCs). Explicit state graph-based algorithms solve this problem in an on-the-fly manner, often providing a counterexample quickly.

Section 2.4 presented the saturation algorithm in detail. In an LTL model checking context, we have to overcome the loss of locality caused by the synchronization with the Büchi automaton describing the property. The goal of Thesis 2 (presented in this chapter and Chapter 5) is to combine the efficiency of saturation with the on-the-fly operation of explicit state model checking. The first step is to utilize saturation to compute the synchronous product on the fly during the state space exploration, which will be presented in this chapter. Then in Chapter 5, the model checking problem is split into smaller tasks according to the iteration of saturation and after each step an incremental model checking query is executed. An efficient, component-wise abstraction technique is used to construct small state graphs tractable by explicit-state algorithms.

The contribution of Thesis 2 is a new hybrid LTL model checking algorithm that *1)* exploits saturation to build the symbolic state space representation of the synchronous product *2)* looks for SCCs on the fly, *3)* incrementally processes the discovered parts of the state space and *4)* uses explicit runs on multiple fine-grained abstractions to avoid unnecessary computations. In the rest of this section, we present an overview of the related work, then Section 4.2 presents the symbolic synchronous product computation.

4.1.1 Related Work

Our algorithm uses a hybrid approach that combines symbolic techniques with abstraction and explicit-state model checking. There are a number of related approaches that solve similar problems based on more or less similar techniques.

Explicit-state LTL model checking computes the graph representation of the synchronous product automaton and uses traditional SCC computation algorithms like the one of Tarjan [Tar72] or more recent ones [HPY97]. It provides a natural way to apply model checking on-the-fly, i. e. continuously during the state space traversal [Cou+91; Ger+95] and answer the model checking question without exploring the full state space in many cases. Explicit-state methods yield the potential of applying various reduction techniques during the traversal such as partial order reduction [Pel98; God96] that is based on cutting redundant orderings of partially ordered actions introduced by the interleaving semantics of the underlying concurrent system models. Partial order reduction is especially efficient for asynchronous, concurrent systems, where state space explosion is a common issue, and can consider LTL properties to preserve information that is important for the evaluation of the property.

Symbolic model checking (discussed in Section 2.3) is used for both CTL (computation tree logic) and LTL model checking. CTL model checking benefits from efficient set manipulation that can be implemented with decision diagram operations (see Section 2.3.3). LTL model checking algorithms were also developed based on decision diagrams and they proved their efficiency [CGH97; STV05]. In these works, the state space and the transition relation of the synchronous product is encoded symbolically, then SCCs satisfying the accepting condition are computed on the synchronous product representation. Symbolic SCC computation based on decision diagrams usually apply greatest fixed point computations on the set of states to compute *SCC hulls* [SRB02]. These approaches typically scale well, and they have been improved considerably due to the extensive research in this area. A particularly interesting approach that also employs abstraction is based on compositional refinement introduced in [Wan+06], which uses techniques similar to the one introduced in Section 5.3.2.

SAT-based methods approach the symbolic model checking problem from a different direction. Traditional SAT-based *bounded model checking* unfolds the state space and the LTL property to a given length (bound), encodes it as a SAT problem and uses solvers to decide the model checking query [Bie+99]. These approaches are incomplete unless the diameter of the state space is reached. In recent years, new algorithms appeared using induction to provide complete and efficient algorithms for model checking of more complex properties, including LTL [SSS00; McM03; Bra+11].

A considerable amount of effort was put in combining symbolic and explicit techniques [BZC99; Dur+11a; HIK04; KP08; STV05]. The motivation is usually to introduce one of the main advantages of explicit approaches into symbolic model checking: the ability to look for SCCs on the fly. Solutions typically include abstracting the state space into sets of states such as in the case of multiple state tableaux [BZC99] or symbolic observation graphs [KP08]. Explicit checks can then be run on the abstraction on the fly to look for potential SCCs.

Our approach builds on these works, as it combines symbolic and explicit techniques too. However, the synchronous product computation is based on new ideas and our approach uses a *series* of fine-grained abstractions instead of a single one to reason about SCCs on the fly. Furthermore, the developed approach employs a novel incremental fixed point computation algorithm to decompose the model checking problem into smaller tasks and incrementally process them.

4.2 Symbolic Computation of the Synchronous Product

As discussed in Section 2.2.5, a crucial point of optimization in LTL model checking is the computation of the synchronous product on the fly during state space generation. In this section, a new algorithm is introduced to efficiently perform this step symbolically.

Formally, given a PTS M and a Büchi automaton A, the task is to directly compute $A_M \cap A$ on the fly using saturation, where A_M is the Büchi automaton accepting the language produced by the state space of M as a Kripke structure (described in Section 2.2.2). Although the result is an automaton, inputs can be omitted from the representation – labels are used only to synchronize the two automata, but are irrelevant in the language emptiness check performed in the next phase of the model checking process.

The algorithm is based on saturation as a state space generation method and reuses the strategy of constrained saturation. The main idea is to decompose the synchronous transitions and to modify constrained saturation to make it compute the possible combinations (see Definition 7). The constraint will serve as a function instead of a set of allowed states, and mechanisms of constrained saturation will be used to evaluate it on states of the model. This approach is presented in Section 4.2.1, then Section 4.2.2 investigates correctness from a formal point of view.

4.2.1 Encoding the Product Automaton

To use saturation, the synchronous product automaton has to be structured to have components and events according to Definition 9 in Section 2.3.1. By the definition of such a structure, saturation can be driven to compute the set of reachable states in the product automaton directly while exploring the state space of the system. Formally, the model of the synchronous product $M_\cap$ has to be defined in the form $M_\cap = (V_\cap, D_\cap, \mathcal{I}_\cap, \mathcal{E}_\cap, \mathcal{N}_\cap)$, also requiring the definition of the variable ordering and the set of events.

Saturation is very sensitive to variable ordering, but the relation between the overall performance (runtime and memory usage) and the ordering is very complex and hard to determine. It is usually advised to place related variables[1] close to each other to enhance locality in the structure of decision diagrams as well. This strategy usually reduces the size of the decision diagram encoding of the set of states. In addition, the representation of events also tends to be smaller. Another thing to consider is the *Top* values of events. A great deal of the power of saturation comes from the ability to apply the partitioned next-state relation locally, thus dividing the fixed point computation into smaller parts. This ability is even more enhanced by caching.

Although it is hard to say how much these values affect performance, one corner case is certainly undesirable: Setting every *Top* value to the index of the highest component, K. In this case, saturation would degrade to a chaining BFS iteration strategy, trying to apply every event on the top level of the decision diagram, effectively flattening the recursive algorithm and degrading cache efficiency in the SATURATE function.

4.2.1.1 Encoding the States

Encoding the states of the product is quite natural in the sense that they are pairs $(\mathbf{s}, q) \in \hat{S} \times Q$, which can be represented as vectors, thus it is possible to encode them in a decision diagram. States of the specification automaton can be represented as a vector in any way from using a single variable

[1]Different definitions of related variables yield different heuristics. For example, variables can be considered related if they are part of the same expression or transitions frequently modify them together.

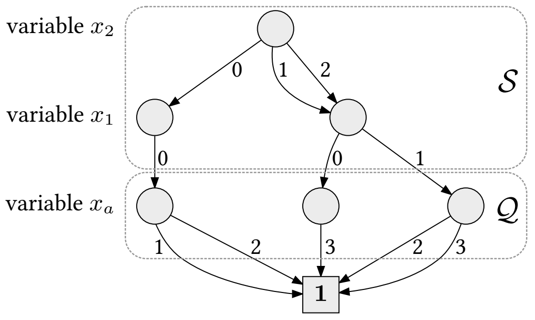


Figure 4.1: Example of a decision diagram encoding of a product state space.

to binary encoding. For now, assume that the state of the specification automaton is encoded in a single variable $x_a = q$ (i. e. it behaves as a single component in $M_\cap$).

The crucial part is the variable ordering, i. e. the order of the original variables $x_1, \ldots, x_K, x_a$ in the state vector of the product. The following heuristic is based on notations and considerations of Section 2.4.1 and assumes that the original model M had a variable ordering already optimized for saturation.

Since events of M and transitions of A are synchronized in $M_\cap$, every event of M will trigger a transition in A, i. e. every event of $M_\cap$ will depend on x_a. Formally, $\forall e \in \mathcal{E}_\cap, x_a \in supp(e)$, and by definition, the value of $\mathit{Top}(e)$ might also change. If the *Top* components of events are not to be changed at all (which is only the most straightforward, but not necessarily the best heuristic), putting x_a to the lowest level is an ideal choice. This way, a state of the product is a vector $(\mathbf{s}, q) = (x_a, s_1, \ldots, s_K)$. Figure 4.1 shows an example of how the encoding decision diagram is structured.

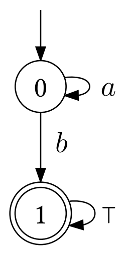
4.2.1.2 Composing the Transition Relation of the Product Automaton

Decomposing the synchronized transitions means that instead of building a single next-state relation encoding the state changes of both M and A, transitions of the model and the automaton are stored and handled separately. The synchronization itself will be done on-the-fly during the state space traversal by our extended constrained saturation-based algorithm, like in the case of prioritized models and EVIDDs in Chapter 3.

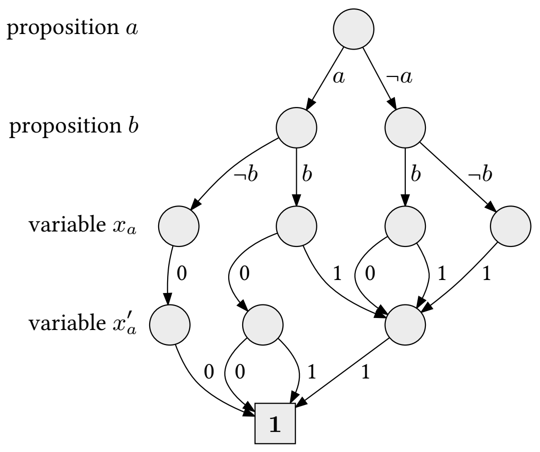
To understand the motivation of the following construct, recall that constrained saturation evaluates a binary function on the states of the system and allows only those that make the function *true* – this is the meaning of traversing a decision diagram encoding a set. Also recall that relations can be interpreted as functions, and they can also be encoded in decision diagrams (this time we will use a hybrid of MDDs and 2K-MDDs). The idea is to use the “function evaluating” ability of constrained saturation to compute the possible state changes of the automaton based on the labeling of the target states of the system (as defined in Definition 7), according to the transition relation of A.

To do this, the transition relation of A is reordered to have the signature $\Delta \subseteq 2^{AP} \times \mathcal{Q} \times \mathcal{Q}$. Assuming an ordering of the atomic propositions of AP where $ind : AP \to \{0, \ldots, |AP| - 1\}$ is the indexing of atomic propositions $p \in AP$, a letter $\alpha \in 2^{AP}$ is encoded as a binary vector $\mathbf{p} \in \mathbb{B}^{|AP|}$, where $\mathbf{p}[ind(p)] = \top \Leftrightarrow p = \top$. For an illustration of such an encoding, observe Figure 4.2b that shows the transition relation of a simple Büchi automaton (presented in Figure 4.2a) as a decision diagram.

It is important to note that the ordering of atomic propositions in $\mathbf{p}$ has to be fixed beforehand and also has to match the order of components that are subjects of propositions. The subject of a



(a) Büchi automaton of $[a \cup b]$.



(b) Constraint encoding the transition relation of the automaton.

Figure 4.2: Minimal Büchi automaton for the LTL formula $[a \cup b]$ and its encoding as a constraint.

proposition is a component whose local state is necessary to evaluate the proposition[2] and is denoted by $Subject(p)$. The valuation of an atomic proposition in terms of the local state i of its subject is $p(i) \in \{0, 1\}$.

With the two transition relations defined separately, a synchronous transition of the product will be computed by applying a transition from the selected $\mathcal{N}_k$ on the state of the system, then choosing a "suitable" transition from Δ. The set of "suitable" transitions can be computed from the function representation of automaton transition relation $\Delta : 2^{AP} \to 2^{Q \times Q}$ by evaluating the atomic propositions on the target state of the system to obtain a letter. Constrained saturation will be used to evaluate the function, with the help of a simple indirection layer evaluating the propositions (see Figure 4.5).

More precisely, as saturation is recursively calling itself and traverses the decision diagram, one of the following can happen. Assume that ℓ is the highest level encoding the automaton ($\ell = 1$ in the pseudocode).

- If the current level belongs to M, i. e. it is above ℓ, the next local transition of $\mathcal{N}$ is used and the constraint evaluates the atomic propositions corresponding to the target local state.
- If the current level is the ℓth, the current constraint node encodes $\Delta(L(\mathbf{s}'))$. For this level and others below it, this relation is used instead of $\mathcal{N}$ to choose the next local transition. The constraint is ignored under this level.
- If the current level is below ℓ (so it belongs to A), the relation $\Delta(L(\mathbf{s}'))$ chosen by α on level ℓ is used to choose the next local transition.

This evaluation step has to be included in both Saturate and RelProd. The modified version of these functions can be seen in Figures 4.3 and 4.4.

The modified saturation algorithm for synchronous product computation is implemented in functions ProdSaturate, ProdRelProd and StepConstraint(Figures 4.3, 4.4 and 4.5, respectively).

StepConstraintis used to navigate through the "constraint" MDD encoding the transition relation of the automaton. It takes as parameters a node c that represents the current state of the evalua-

[2]Note that the current version of the algorithm does not support the comparison of variables in different components, so the subject of an atomic proposition is always a single component. In case of bounded variables with the same domain, this restriction can be circumvented by comparing both of them to the same constant value for all possible values in the domain.

tion and two indices i and k, the former encoding the reached local state of the model and the latter identifying the current level (or component).

The function performs a simple evaluation. If the current level is the lowest (i. e. belongs to the automaton), the function does nothing, but returns c (that may be $\mathbf{0}$ or an MDD encoding state transitions of the automaton). Otherwise atomic propositions belonging to level k are evaluated on local state i in the predefined order and the constraint is navigated along the corresponding edges. Note that this navigation may consist of zero or more steps, depending on the number of propositions belonging to level k.

Functions ProdSaturate and ProdRelProd are very similar to Saturate and RelProd. There are essentially two differences. First, where constrained saturation navigates the constraint by getting a child of the current node, the modified version uses StepConstraintto compute the next constraint node (note that the level of c is now unknown). Secondly, when ProdRelProd reaches level 1, the node encoding the next-state relation of the model should be $\mathbf{1}$ – this is the point where the constraint is used as the transition relation of the automaton (line 2).

Formally, the algorithm applies the following set of transitions: $\{((\mathbf{s},q),(\mathbf{s}',q')) \mid (\mathbf{s},\mathbf{s}') \in \mathcal{N},(q,q') \in \Delta\left(L(\mathbf{s}')\right)\}$, i. e. the first part of a transition (originally belonging to the model) takes the model into a state $\mathbf{s}'$, whose labeling is read to choose the second part of the transition (originally belonging to the automaton). $\Delta\left(L(\mathbf{s}')\right)$ can be seen as a partitioning of the transition relation of the automaton based on the letters that transitions read.

4.2.2 Correctness and Efficiency

In order to prove the correctness of the algorithm, it is necessary to show that every transition of the product can be simulated by the decomposition-based approach, and no false transitions are introduced by the construct.

Theorem 1 (Correctness of decomposition-based product computation) Given the transition relation $\Delta_\cap$ of the product automaton $A_M \cap A$, the next-state relation $\mathcal{N}$ of M and the transition relation Δ of A, every transition of $\Delta_\cap$ can be simulated by the decomposition-based product computation algorithm and vice versa. ▪

Proof *The definition of the transition relation of the product (without the input letters) is* $\Delta_\cap = \{((\mathbf{s},q),(\mathbf{s}',q')) \mid (\mathbf{s},\mathbf{s}') \in \mathcal{N},(q,\alpha,q') \in \Delta,\alpha = L(\mathbf{s}')\}$. *The modified constrained saturation applies the transitions* $\{((\mathbf{s},q),(\mathbf{s}',q')) \mid (\mathbf{s},\mathbf{s}') \in \mathcal{N},(q,q') \in \Delta\left(L(\mathbf{s}')\right)\}$. *By definition,* $\Delta(\alpha) = \{(q,q') \mid (q,\alpha,q') \in \Delta\}$, *so by substituting* $\alpha = L(\mathbf{s}')$ *the equivalence follows.*

As the proof suggests, the algorithm directly computes the synchronous transitions on the fly without actually storing them. This way, no additional storage is required above the representation of the separate relations, and the computational overhead is also negligible compared to traditional constrained saturation.

```
     input   : $s_k$, $c$ : node
     // $s_k$: node to be saturated,
     // $c$: constraint node
     output  : node
   1 if $s_k = \mathbf{1}$ then return 1
   2 $n_{2k} \leftarrow \mathcal{N}_k$ as decision diagram
   3 Return result from cache if possible
   4 $t_k \leftarrow$ new $Node_k$
   5 for each $i \in \mathcal{S}_k : s_k[i] \neq \mathbf{0}$ do
 * 6     $c' \leftarrow StepConstraint(c, i, k)$;
   7     if $c' \neq \mathbf{0}$ then $t_k[i] \leftarrow$ ProdSaturate$(s_k[i], c')$
   8     else $t_k[i] \leftarrow s_k[i]$                                   // no steps allowed
   9 end
  10 repeat
  11     for each $s_k[i] \neq \mathbf{0} \wedge n_{2k}[i][i'] \neq \mathbf{0}$ do
 *12         $c' \leftarrow StepConstraint(c, i', k)$
  13         if $c' \neq \mathbf{0}$ then $t_k[i'] \leftarrow (t_k[i'] \cup$ ProdRelProd$(t_k[i], c', n_{2k}[i][i']))$
  14     end
  15 until $t_k$ unchanged
  16 $t_k \leftarrow PutInUniqueTable(t_k)$
  17 Put inputs and results in cache
  18 return $t_k$
```

Figure 4.3: ProdSaturate

```
     input   : $s_k, c, n_{2k}$ : node
     // $s_k$: node to be saturated,
     // $c$: constraint node,
     // $n_{2k}$: next-state node
     output  : node
   1 if $s_k = \mathbf{1} \wedge n_{2k} = \mathbf{1}$ then return 1
 * 2 if $k = 1$ then $n_{2k} \leftarrow c$                          // transitions of automaton
   3 Return result from cache if possible
   4 $t_k \leftarrow$ new $Node_k$
   5 for each $s_k[i] \neq \mathbf{0} \wedge n_{2k}[i][i'] \neq \mathbf{0}$ do
 * 6     $c' \leftarrow StepConstraint(c, i', k)$;
   7     if $c' \neq \mathbf{0}$ then $t_k[i'] \leftarrow (t_k[i'] \cup$ ProdRelProd$(s_k[i], c', n_{2k}[i][i'])$
   8 end
   9 $t_k \leftarrow$ ProdSaturate$(PutInUniqueTable(t_k), c)$
  10 Put inputs and results in cache
  11 return $t_k$
```

Figure 4.4: ProdRelProd

```
     input   : $c$ : node $i$, $k$: index
     // $c$: constraint node,
     // $i$: index of local state
     // $k$: level of component
     output  : node
   1 if $k \leq 1$ then return $c$                      // level of automaton, do nothing
   2 foreach $p \in AP, Subject(p) = k$ do
   3     $c \leftarrow c[p(i)]$                                    // evaluate $p$ on $i$
   4 end
   5 return $c$
```

Figure 4.5: StepConstraint

Chapter 5

Saturation-Based Detection of Strongly Connected Components for LTL Model Checking

5.1 Introduction

Chapter 4 introduced a new algorithm to build the product state space encoded by decision diagrams on the fly by using saturation. In this chapter, we will achieve on-the-fly detection of SCCs by running searches over the discovered state space continuously during state space generation. In order to reduce the overhead of these searches, we present a new incremental fixed point algorithm that considers newly discovered parts of the state space when computing the SCCs. This approach relies on the component-wise structure of asynchronous systems and incremental fixed-point computation is driven by the ordering of the components.

Abstraction is a key technique in the verification of complex systems, where the choice of the applied abstraction function determining the information to be hidden is very important. The computational cost of the abstraction is also significant: the more complex the chosen abstraction function is, the less efficient the model checking procedure might become. However, the computational investment of having better abstraction can pay off in decreasing the verification costs. Choosing the proper abstraction is difficult, many attempts exist targeting this problem, e. g. in [Cla+00; Hen+02; Wan+06]. While the incremental fixed point algorithm specializes on *finding* an SCC, we will use abstraction in a complementary algorithm that maintains various abstractions of the state space to perform explicit searches in order to inductively prove the *absence* of SCCs.

This chapter is structured as follows. Section 5.2 details the new symbolic SCC computation algorithm. The efficiency of SCC computation is further enhanced by various heuristics and abstractions, discussed in Section 5.3. The proposed new approach to LTL model checking, including the results presented in Chapter 4, is evaluated and compared to three other tools in Section 5.4. Finally, the work is concluded in Section 5.5.

5.1.1 Main Concepts

There are simple and powerful algorithms for SCC detection in the explicit case, where the state space is represented by a graph that can be traversed freely [Tar72; HPY97]. In a symbolic setting, set operations can be used to compute an SCC-hull as a greatest fixed point in the state space [SRB02]. To introduce the advantage of on-the-fly SCC detection and early termination of explicit techniques into symbolic algorithms, hybrid approaches using abstraction have also spread [HIK04; KP08; Dur+11a] (implemented in e. g. ITS-tools[1]). To further enhance the power of these approaches, both symbolic and explicit methods will be introduced in the following two sections, carefully designed to work together in a symbolic setting.

The symbolic algorithm will be a variant of traditional SCC-hull computation schemes, but optimized to process the state space *incrementally* during the exploration. On the verge of symbolic and explicit, saturation will be enhanced to collect recurring states, states that are visited more than once during the exploration and thus can indicate SCCs. Finally, a cheap abstraction considering decision diagram nodes will also be employed to use explicit algorithms to quickly reason about SCCs in the state space without actually trying to find them. The last two techniques are capable of making the overhead of on-the-fly model checking almost disappear.

Algorithms of this chapter are orthogonal to the algorithm of Chapter 4. Every one of them assumes that a model is given as a PTS, no matter if it is a product automaton or anything else. Consequently, the devised methods can be used in settings other than LTL model checking, although in the thesis, the focus is on this particular application.

Before presenting the algorithms, a seemingly trivial but fundamental lemma has to be declared. This observation will serve as the basis of the incremental symbolic SCC detection algorithm as well as the explicit and abstraction techniques of the next section.

Lemma 3 (Lemma of state space partitions) Given a decision diagram node n on level $\mathit{lvl}(n) = k$, any path of $(S(n), \mathcal{N}_{\leq k})$ as a directed graph (the state graph of submodel $M_{\leq k}$ restricted to states encoded by n) that is not present in $(S_{[i]}(n), \mathcal{N}_{<k})$ (the state graph of submodel $M_{<k}$ restricted to states encoded by child nodes) contains at least one transition from $\mathcal{N}_k$. ▪

An implication of the lemma is that there is no way to traverse the boundaries of $S_{[i]}(n)$ without using at least one transition from $\mathcal{N}_k$, since the state spaces $(S_{[i]}(n), \mathcal{N}_{<k})$ are *disjoint* (they differ in the local state of component k).

5.2 Symbolic SCC Computation

As seen in Section 2.2.5, the LTL model checking problem can be reduced to checking language emptiness, which in turn reduces to the problem of finding fair SCCs in the product state space. This is why devising efficient SCC detection algorithms is especially important here. The algorithm presented in this section is correct and complete. However, various heuristics can be used to further improve its efficiency. These extensions will be discussed later in Section 5.3.

5.2.1 Related Work: SCC Computation with Saturation

Saturation-based SCC computation has first been proposed by [ZC11]. The two implemented algorithms are that of Xie and Beerel (XB) and the Transitive Closure (TC) algorithm. Both of them differ from SCC-hull algorithms, because they aim to compute exactly those states that belong to an SCC.

The main idea of the XB algorithm is to compute the set of forward and backward reachable states from a randomly picked seed state – their intersection gives an SCC. After removing the states of this SCC, the procedure is repeated until no states are left.

[1] https://lip6.github.io/ITSTools-web/

The TC method works by building the relation of reachability between reachable states, i. e. the transitive closure of the next-state relation. This relation can then be used to identify states in SCCs in a fully symbolic way by collecting all states that have a self loop in the transitive closure.

While the XB algorithm is usually much faster, it does not scale well with the number of SCCs in the state space. TC, on the other hand, does not enumerate the strongly connected components, but encodes them symbolically.

These algorithms differ from the approach presented here in that they both compute exact SCCs of an already explored state space, whereas the we compute SCC hulls and do it on the fly, during state space exploration. Nevertheless, due to the ability to reuse the caches of other runs of saturation, these algorithms can be very efficient in computing an exact counterexample detected by the algorithm presented in Sections 5.2.2–5.2.2.3.

5.2.2 Incremental Symbolic SCC Computation

In the presented model checking algorithm, on-the-fly SCC detection is intended to be achieved by running SCC computations *frequently* during the state space generation. Certainly, this strategy would mean a massive overhead if repetition of work would not be addressed, so the main requirement towards the design of such an approach is incrementality.

The context of the algorithm is the following. When saturation processes the state space (see Section 4), SCC detection will be run whenever a node n becomes saturated. Processing a saturated node has the advantage of handling a set of (partial) states $S(n)$ that is closed with regard to events independent from higher levels ($\mathcal{N}_{\leq k}$). This means that the set will not change anymore during the exploration, i. e. each closed set has to be processed only once.

Even though a set with its related events will be processed only once, the recursive definition of saturation will cause such sets to appear again as subsets of larger sets encoded by the parent node in the decision diagram (see Section 2.3.3). Due to this, the algorithm has to be able to distinguish parts of the state space already processed (these are $(S_{[i]}(n), \mathcal{N}_{<k})$ encoded by the children nodes) and focus only on new opportunities gained by processing the current node. Since $S(n) = \bigcup_{i \in D(x_k} S_{[i]}(n)$ and $\mathcal{N}_{\leq k} = \mathcal{N}_{<k} \cup \mathcal{N}_k$ of which $S_{[i]}(n)$ and $\mathcal{N}_{<k}$ are already processed, new elements are in $\mathcal{N}_k$. This is exactly the base idea of the algorithm: in each run, look for only those SCCs that contain at least one transition from $\mathcal{N}_k$.

5.2.2.1 Elementary Steps of the Fixed Point Computation

The idea can be implemented by discarding transitions from a set of “new” transitions $\mathcal{N}_{new}$ – those that cannot be closed to form a loop. In other words, a transition is discarded if its *source state* cannot be reached from its *target state*. In SCC-hull algorithms [SRB02], sets of states are processed iteratively to eventually get rid of “bad” states (dead-end states, for example) by computing a greatest fixed point of some function on the original set. Now, a set of transitions has to be processed and the function is the following ($\mathcal{N}$ is the set of all transitions, i. e. both “old” and “new” transitions):

$$f(Z) = \{(\mathrm{s}_1, \mathrm{s}'_1) \mid (\mathrm{s}_1, \mathrm{s}'_1) \in Z, \exists (\mathrm{s}_2, \mathrm{s}'_2) \in Z, \mathrm{s}_1 \in \mathcal{N}^*(\mathrm{s}'_2)\}$$

The formal goal of the proposed SCC detection algorithm is to compute the greatest fixed point of f as $Z_\Theta = f(Z_\Theta) \cap Z_\Theta \subseteq \mathcal{N}_{new}$, i. e. transitions of $\mathcal{N}_{new}$ that are fireable on some path after another transition of the set has fired. The intuition is that in in an SCC, every transition may be fired after some other. The following lemma justifies the definition of function f and will prove the correctness and completeness of an incremental step of the SCC detection algorithm.

Lemma 4 (Correctness and completeness) Given a transition system $(\mathcal{S}, \mathcal{N})$ and a set of "new" transitions $\mathcal{N}_{new} \subseteq \mathcal{N}$, the fixed point Z_Θ is *empty* iff $(\mathcal{S}, \mathcal{N})$ does not contain any SCC with transitions from $\mathcal{N}_{new}$. ▪

Proof *The two directions are proven separately.*

$(\rightarrow)$*: Indirect proof. Suppose there is a strongly connected component* Θ *in the transitions system that contains at least one transition from* $\mathcal{N}_{new}$*, let this be* $(\mathrm{s}, \mathrm{s}')$*. Also suppose that the fixed point* Z_Θ *is empty. Consider* $f(\{(\mathrm{s}, \mathrm{s}')\})$*. By the definition of a strongly connected component, every state of* Θ *is reachable from every other state of* Θ*, so* s *is also reachable from* s'*. Reachability in terms of* $\mathcal{N}$ *is defined by inclusion in the set of states* $\mathcal{N}^*(\mathrm{s}')$*, so* $(\mathrm{s}, \mathrm{s}') \in f(\{(\mathrm{s}, \mathrm{s}')\})$ *can be concluded that contradicts the assumption of* Z_Θ *being empty. Note that because of the transition* $(\mathrm{s}, \mathrm{s}')$*,* Θ *is a real (but not necessarily nontrivial) SCC even if* $\mathrm{s} = \mathrm{s}'$ *because of the self-loop.*

$(\leftarrow)$*: The other direction is also proved indirectly. Suppose there is no strongly connected component in the transition system that contains at least one transition from* $\mathcal{N}_{new}$*. Also suppose that the fixed point is nonempty. Take a transition* $\nu_1 = (\mathrm{s}_1, \mathrm{s}'_1)$ *from* Z_Θ*. Since it is in the fixed point, its source state* s_1 *must be reachable from the target state* s'_2 *of some other transition* $\nu_2 = (\mathrm{s}_2, \mathrm{s}'_2)$*. Now consider this transition and repeat the process. Since* Z_Θ *is finite, at some point, the transition* ν_i *will be the same as some transition before:* $\nu_i = \nu_j = (\mathrm{s}, \mathrm{s}')$*, where* $j < i$*. At each repetition, the source state of previous transitions were reachable from the target state of the current transition, so* $\mathrm{s}_j = \mathrm{s}$ *is reachable from* $\mathrm{s}'_i = \mathrm{s}'$*. Since* s' *is obviously reachable from* s *through* ν*, they must be in an SCC, which leads to a contradiction.*

5.2.2.2 Incremental Steps of the Fixed Point Computation

An incremental step has to compute the fixed point Z_Θ. This is implemented by the function DetectSCC that can be seen in Figure 5.2. Checking reachability is performed using saturation, enabling the algorithm to *reuse caches* in the SCC detection phase as well, just like the decision diagram structures built during the state space exploration (through the shared unique table).

Compared to the definition of f, the implementation is slightly different. Instead of handling a set of transitions, the implementation reduces the problem to sets of states by considering the source states and target states of every transition in $\mathcal{N}_{new}$. The sets of source and target states are denoted by $\mathcal{S}^-$ and $\mathcal{S}^+$ respectively. Furthermore, a set of states $\mathcal{S}$ (typically the set of states discovered so far) is also input to DetectSCC to constrain SCCs – during the state space generation (especially if saturation is used), $\mathcal{N}$ may contain transitions that are not in $\mathcal{S} \times \mathcal{S}$ (e. g. they will be reached after firing an event from $\mathcal{E}_{>k}$ or not reachable at all).

The core of DetectSCC is the *filtering loop* (lines 3–6), where function f is implemented and performed iteratively until no more changes occur. In each iteration, the sets $\mathcal{S}^-$ (source states) and $\mathcal{S}^+$ (target states) are filtered:

1. Elements of $\mathcal{S}^-$ that are not reachable from $\mathcal{S}^+$ are removed.
2. Elements of $\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ in one step through $\mathcal{N}_{new}$ are removed, ensuring that $\mathcal{S}^-$ and $\mathcal{S}^+$ always contain exactly the source and target states of the remaining transitions.

Lemma 4 still holds if Z_Θ is approximated by $\mathcal{S}^-$ and $\mathcal{S}^+$: transitions are removed from Z when their source states are removed from $\mathcal{S}^-$ in step 1, while target states are removed from $\mathcal{S}^+$ when all of their corresponding transitions got removed from Z to adjust the approximation in step 2. Note that $\mathcal{S}^-$ and $\mathcal{S}^+$ will always be both empty or both nonempty at the fixed point, because a transition has to have a source *and* a target state. However, the iteration can be stopped one step before – if any of them becomes empty, the next step will discard every state from the other one as well.

The number of iterations in the filtering loop has an upper bound of $\mathcal{O}(|\mathcal{N}_{new}|)$, since in every step, at least one transition is discarded from the set. Methods to make the initial set of "new" transitions smaller and thus reduce the number of required steps are discussed in Sections 5.3.3.1 and 5.3.3.3.

An incremental step will always assume that there is no SCC in the transition system that *does not contain* a transition from $\mathcal{N}_{new}$ (otherwise the on-the-fly model checking algorithm would have already been terminated). With this, the completeness of the algorithm depends on the strategy of applying the incremental steps.

5.2.2.3 On-the-fly Search Using the Incremental Steps

The last design question is how and when to call DetectSCC. The chosen design is to call DetectSCC whenever a node n becomes saturated (marked by a circle in Figure 5.6, presenting the final pseudocode of on-the-fly SCC detection). The set of states to constrain the search is $S(n)$, the set of all transitions is $\mathcal{N}_{\leq k}$ and the set of new transitions is $\mathcal{N}_k$ (it is trivial that $\mathcal{N}_k \subseteq \mathcal{N}_{\leq k}$).

In the general case (and in breadth-first style strategies), the whole next-state relation of a model can be partitioned by the traversal strategy, for example, a breadth-first exploration would partition the transitions into "layers" based on their distance from the initial state. If DetectSCC is called on each partition as new transitions, Lemma 4 will imply that the algorithm is correct and complete, i. e. it finds an SCC exactly if there exists one.

In saturation, however, the exploration is recursive, so calling DetectSCC after a node is saturated does not fall into the above case. Proving that the algorithm is complete can thus be performed inductively.

Theorem 2 (Completeness of incremental SCC detection) Calling DetectSCC during state space generation every time a node becomes saturated gives a complete algorithm for deciding if there exists an SCC in a state space, i. e. in at least one call, the fixed point will not be empty iff there exists an SCC in the state space. ▪

Proof *Inductive proof. It is trivial that the empty state spaces encoded by terminal nodes do not contain any SCC. Assume the children of a decision diagram node n together with their related transitions $\mathcal{N}_{<k}$ (that is, $(S(n[i]), \mathcal{N}_{<k})$) do not contain SCCs either. Proving that the fixed point will be nonempty iff there exists an SCC in $(S(n), \mathcal{N}_{\leq k})$ would imply that when the root node is saturated and the algorithm stops, the statement of the theorem would hold. The inductive hypothesis directly follows from Lemmas 3 and 4. If there exists an SCC, it must contain a path that is not present in $(S(n[i]), \mathcal{N}_{<k})$ (otherwise $(S(n[i]), \mathcal{N}_{<k})$ would also contain the SCC), so according to Lemma 3 it will contain at least one transition from $\mathcal{N}_k$. This would cause the fixed point to be nonempty (Lemma 4).*

5.2.3 Extensions to Support Fair SCCs

When looking for fair SCCs, i. e. SCCs containing at least one state from a set of states $\mathcal{F}$, the algorithm can be extended to involve $\mathcal{F}$ as the third set in the filtering loop. Operations performed in the cycle are then the following (the main idea is illustrated in Figure 5.1):

- Elements of $\mathcal{F}$ that are not reachable from $\mathcal{S}^+$ are removed.
- Elements of $\mathcal{S}^-$ that are not reachable from $\mathcal{F}$ are removed.
- Elements of $\mathcal{S}^+$ that are not reachable from $\mathcal{S}^-$ in one step through $\mathcal{N}_{new}$ are removed, ensuring that $\mathcal{S}^-$ and $\mathcal{S}^+$ always contain exactly the source and target states of the remaining transitions.

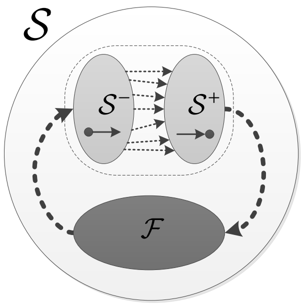


Figure 5.1: Illustration of Figure 5.2 extended to look for fair SCCs.

```
input   : $\mathcal{S}, \mathcal{N}, \mathcal{N}_{new}$ : set
// $\mathcal{S}$: set of states,
// $\mathcal{N}, \mathcal{N}_{new}$: set of transitions
output  : bool
1 $\mathcal{S}^- \leftarrow \mathcal{N}_{new}^{-1}(\mathcal{S})$;   $\mathcal{S}^+ \leftarrow \mathcal{N}_{new}(\mathcal{S}^-)$
2 if $\mathcal{S}^+ = \emptyset$ then return false
3 repeat
4     $\mathcal{S}^- \leftarrow \mathcal{S}^- \cap \mathcal{N}^*(\mathcal{S}^+)$
5     $\mathcal{S}^+ \leftarrow \mathcal{S}^+ \cap \mathcal{N}_{new}(\mathcal{S}^-)$
6 until $\mathcal{S}^+$ and $\mathcal{S}^-$ unchanged
7 return $\mathcal{S}^- \neq \emptyset \wedge \mathcal{S}^+ \neq \emptyset$
```

Figure 5.2: DetectSCC

The first two operations ensure that if a transition of $\mathcal{N}_{new}$ is in an unfair SCC (i. e. does not contain any state from $\mathcal{F}$), it is removed from the fixed point. Note that even accepting trivial SCCs (a single state of $\mathcal{F}$ with a transitions of $\mathcal{N}_{new}$ as a self loop) are found this way, because a state is by definition reachable from itself (i. e. reachability as a relation is reflexive).

When looking for accepting SCCs during LTL model checking, $\mathcal{F}$ is the set of accepting states. Supporting multiple acceptance sets to directly use more complex automata (e. g. generalized Büchi automata) in the model checking algorithm is subject of future work.

5.3 SCC Computation Made Smart

The SCC computation algorithm presented before is efficient and generally applicable. However, by considering that the goal of any on-the-fly model checking algorithm is to terminate when the first counterexample is found, it is easy to see that at most one SCC detection call is sufficient. In this section, we extend our SCC computation algorithm by adding various methods to *prove the absence* of SCCs and thus prevent unnecessary symbolic fixed-point computations.

When looking for accepting SCCs, checking the absence of accepting states is a usual optimization in similar algorithms, for example in the abstraction refinement approach presented in [Wan+06]. The contribution of this thesis goes two steps further. Section 5.3.1 introduces the use of recurring states and a specialized algorithm to compute them, while Section 5.3.2 presents new component-wise abstraction techniques tailored to decision diagrams and saturation that allow the use of explicit algorithms directly to reason about the presence or absence of SCCs.

The heuristic using recurring states is based on the observation that if no states were seen multiple times during the saturation, then no loop can exist in the state space. The use of abstraction exploits the fact that if an over-approximation of the state space does not contain an SCC, then the state space cannot contain any either. A suitably small abstraction can be used with efficient explicit algorithms that are far cheaper than an actual symbolic detection step.

The new SCC computation workflow is summarized in Figure 5.3. After presenting the building blocks, Sections 5.3.3 and 5.3.4 will present the complete SCC algorithm extended with these improvements.

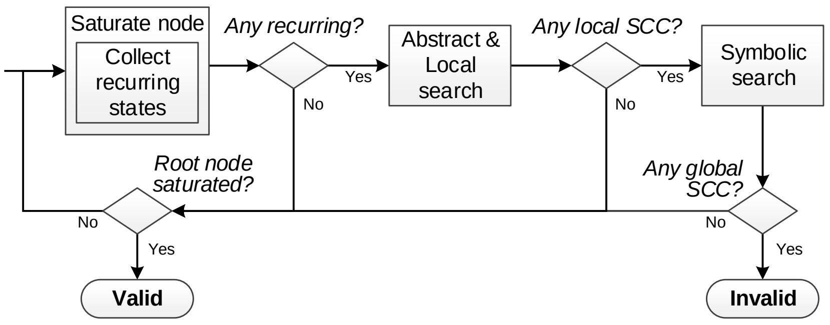


Figure 5.3: SCC computation workflow extended with heuristics.

5.3.1 Recurring States Heuristics

Recurring states are those that have already been discovered before reaching them again during state space exploration. To precisely define them, let a concrete *exploration* ϵ of a fully reachable, connected state graph $(\mathcal{S}, \mathcal{N})$ with initial states $\mathcal{I}$ be a sequence of subsets of $\mathcal{S}$, where each element of the sequence contains states discovered in that step: $\epsilon \in (2^{\mathcal{S}})^*$ such that $\epsilon(0) = \mathcal{I}$ and $\epsilon(i+1) \subseteq \mathcal{N}(\mathcal{S}_i)$ for every $0 \leq i < |\epsilon|$, where $\mathcal{S}_i = \bigcup_{0 \leq j \leq i} \epsilon(j)$. An exploration is *full* if $\mathcal{S}_{|\epsilon|} = \mathcal{S}$ and the exploration algorithm considered every enabled transition in $\mathcal{N}$.

Definition 32 (Recurring states) Given an exploration $\epsilon \in (2^{\mathcal{S}})^*$, the set of recurring states in each step i is $\mathcal{R}_i = \mathcal{S}_i \cap \epsilon(i+1)$, where $\mathcal{S}_i = \bigcup_{0 \leq j \leq i} \epsilon(j)$. ▪

Many explicit algorithms rely on recurring states as indicators of SCCs (for example [Tar72] and [HPY97]). Indeed, they are candidates of being in an SCC, since (apart from the trivially avoidable case of applying the same transitions multiple times) there are only two cases in which they can appear: if the exploration reached them on parallel paths, or a previous state of a path is reached again, i. e. *a loop is found.* Using the following proposition, recurring states can be used to reason about SCCs.

Proposition 2 Given a state space containing at least one SCC, any full exploration will yield at least one state of each SCC as a recurring state. ▪

Using the proposition as an indirect proof, recurring states offer a cheap way to distinguish situations where there is no chance of finding an SCC – situations that often arise during an on-the-fly search.

5.3.1.1 On-the-fly Collection of Recurring States

Since a basic step in saturation is performed by applying some $\mathcal{N}_e$ (i. e. computing the relational product of a set of states and $\mathcal{N}_e$), recurring states have to be collected in this granularity as well. In this context, the desired output of the function RelProd would be two sets of states: the result of the relational product and the set of recurring states.

Instead of performing the costly intersection at the end of the function, the collection of recurring states can be done on the fly. In general, constrained saturation did exactly the same, as it only allowed steps that stay in a certain set of states. This set is now the set of "old" states, i. e. those that were passed to RelProd.

Figure 5.7 shows the implementation of RelProd with the collection of recurring states, corresponding lines marked by asterisks. The function basically performs a constrained and an unconstrained saturation simultaneously, gathering the results to two separate sets (decision diagrams). An important note is that contrary to normal constrained saturation, the recursion occurs even if the constraint would not allow the step, because the main functionality is the computation of the relational product.

In each Saturate call (Figure 5.6 shows the extended version, where Saturate is called SccSaturate), all recurring states produced by transitions of a given $\mathcal{N}_k$ are collected separately as the union of sets returned by RelProd (called SccRelProd). Recurring states encountered by applying lower level events during recursive Saturate calls are processed locally and will not be considered any further. The arguments of RelProd are the original node, the next-state node and the node that represents the relevant part of the old states (with which the results of the relational product will be merged).

It is important to note that this mechanism is useful only if the inputs and results of SccRelProd are cached. Otherwise, repeating a previous call would by definition recognize every target state as a recurring state, since they have already been reached by that previous call.

5.3.2 Using Abstractions to Reason About Emptiness

Hybrid model checking algorithms usually use symbolic encoding to process huge state spaces, while introducing clever abstraction techniques to produce an abstract model on which explicit graph algorithms can be used. In this context, the goal of abstraction is to reduce the size of a system's state space while preserving certain properties, such as the presence or absence of SCCs. The purpose of abstractions is no different in this work, they are used to reason about SCCs. However, unlike in most approaches in this domain, multiple abstractions are used, ordered in a hierarchy matching the structure of the underlying decision diagram to build an inductive proof about strongly connected components of the state space.

In a symbolic setting, components of the model provide a convenient basis for abstraction (see e. g. projection in Section 2.3.2 or submodel in Section 2.4.2). In LTL model checking, it is usual to use the Büchi automaton or its observable language to group states and build an abstraction from these aggregates.

Abstraction based on components. An extensive approach to using abstraction in SCC computation has been proposed in [Wan+06]. By defining a lattice of abstractions based on one or more components of the model, the paper presents strategies of using some of the abstractions to discard uninteresting parts of the state space and search in relevant components.

Each abstraction is obtained by projecting the state space to some of the components, keeping only those transitions that do not affect other components (in this sense, they are similar to the must abstraction of Section 5.3.2.1). Searching for SCCs in the abstract graphs can prove that no SCCs exist within the selected components. If an SCC is found, the algorithm also looks for accepting states, which offers another way of discarding irrelevant components. Completeness is achieved by refining the abstraction until an SCC is found or the full state space is searched.

Abstraction based on the automaton. A common way of abstracting the state space is based on the automaton describing the desired property. Such techniques include symbolic observation graph (SOG) [HIK04; KP08], its extension, symbolic observation product (SOP) [Dur+11a], and self-loop aggregation product (SLAP) [Dur+11b].

Symbolic observation graphs are aggregated Kripke structures: each state of the SOG is a set of states of the original model. Consecutive states are grouped by observable atomic propositions, i. e. two states are considered equal if the satisfy the same atomic propositions of the LTL formula. An improvement is symbolic observation product, which is based on the observation that as the property automaton progresses, the set of relevant atomic propositions decreases, allowing a more aggressive grouping (i. e. two states are considered equal if they satisfy the same atomic propositions currently read by the automaton). Furthermore, SOP can substitute the computation of the product automaton, because it is in itself a tableau representing the system in terms of the property. The drawback of this approach is that it requires a globally stuttering property (i. e. operator X cannot be used).

Self-loop aggregation product works with every LTL formula. SLAP aggregates consecutive states by observing self loops of the Büchi automaton: states are aggregated if their labels satisfy the condition of self-loops, i. e. the automaton does not change its state when the system does (i. e. this transition of the system is invisible to the Büchi automaton).

5.3.2.1 Simple abstractions

In addition to selecting the base component, there are multiple ways to define an abstraction in terms of transitions. To illustrate this, two simple abstractions are presented before introducing a new approach of using the structure of a decision diagram to define a more powerful abstraction.

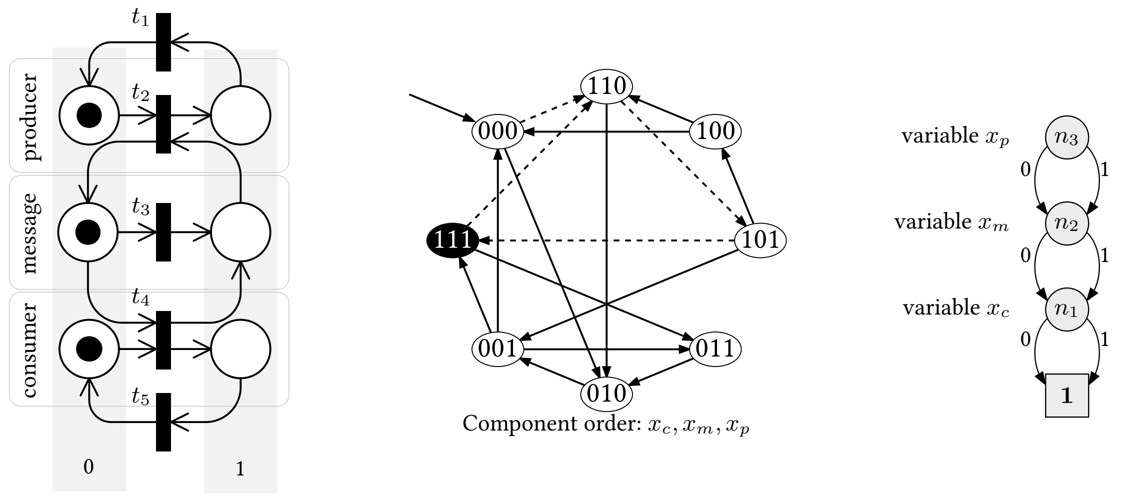
Using abstractions to answer binary decisions has two potential goals. One can create an under-approximating abstraction that can say a definite *yes* (these are *must abstractions*, e. g. as defined in Section 2.3.2), or an over-approximating one that can say a definite *no* (these are *may abstractions*). To construct an abstraction based on a single component of the system, we can use *submodel* as must abstraction and *projection* as may abstraction.

In a submodel, the must abstraction of transitions is defined to keep only those transitions that correspond to events fully within the support of the chosen component. We will denote the state graph of a submodel for component k as $\mathcal{A}_k^{\exists} = (\mathcal{S}_k, \mathcal{N}_k^{\exists})$. In a projection, may abstraction preserves every local transition, but omits the synchronization between components (i. e. assumes that if a transition is enabled in component k, it is globally enabled). We will denote the state graph of a submodel for component k as $\mathcal{A}_k^{\forall} = (\mathcal{S}_k, \mathcal{N}_k^{\forall})$.

Due to these definitions, it is sometimes possible to reason about the presence or absence of global SCCs. If there is an SCC in a single must abstraction, it is the direct representation of one or more SCCs of the global state space. Complementary, if there is no SCC in the may abstraction of *any* component, then the global state space cannot contain any SCC either.

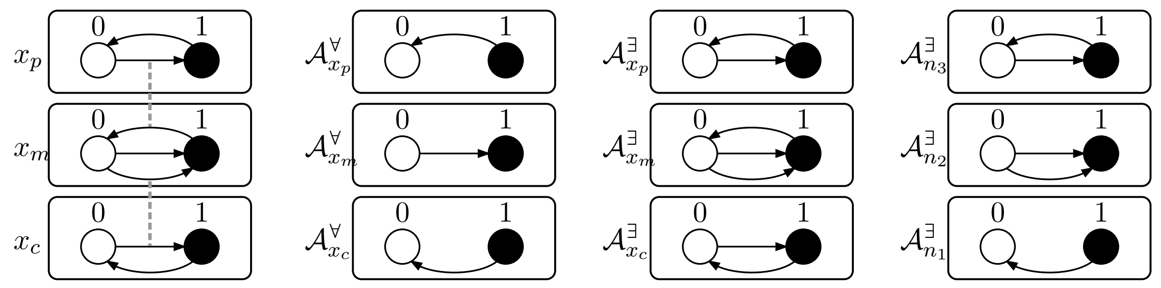
These abstractions usually yield small state graphs that can be represented explicitly. Running linear-time explicit algorithms on them gives a very cheap opportunity to possibly prove or refute the presence of SCCs before symbolic methods are used. Moreover, the definition of may and must abstractions implies $\mathcal{N}_k^{\forall} \subseteq \mathcal{N}_k^{\exists}$, so running the explicit SCC computation on a may abstraction and checking if every transition of a possible SCC is in $\mathcal{N}_k^{\forall}$ effectively considers both cases at the same time.

Example 1 *As an example, observe Figure 5.4. Figure 5.4a illustrates the Petri net model of a producer-consumer system where the message has a timeout if the consumer is too slow. The model is divided into three components, with their state variables (x_c for the consumer, x_m for the message and x_p for the producer) having two potential values: 0 if their token is on the left place and 1 if it is on the right. Figures 5.4b and 5.4c show the state space of the model as an explicit state graph and an MDD, with variable ordering x_c, x_m, x_p.*



(a) Petri net model. (b) State space. (c) State space MDD.

Figure 5.4: Producer-consumer model with non-deterministic buffer.



x_p
x_m
x_c
$\mathcal{A}^{\forall}_{x_p}$
$\mathcal{A}^{\forall}_{x_m}$
$\mathcal{A}^{\forall}_{x_c}$
$\mathcal{A}^{\exists}_{x_p}$
$\mathcal{A}^{\exists}_{x_m}$
$\mathcal{A}^{\exists}_{x_c}$
$\mathcal{A}^{\exists}_{n_3}$
$\mathcal{A}^{\exists}_{n_2}$
$\mathcal{A}^{\exists}_{n_1}$
0
1

(a) Original (b) Must (c) May (d) Node-wise

Figure 5.5: The effect of the abstractions to the transitions

State transitions of the system are shown in Figure 5.5a, with connected arcs representing a single transition affecting multiple components. Assume that every transition of the Petri net defines an event in the model. In this case, every state transition belongs to a separate event. Events affecting multiple components can be regarded as synchronization constraints between local transitions. *Abstractions can be acquired by removing synchronizations and local transitions. Figures 5.5b and 5.5c depict the transitions transformed by must and may abstractions, respectively, for every component. If the goal is to find an SCC containing the state where only the places on the right of the Petri net are marked (depicted as a black state in Figure 5.4b), none of the simple abstractions of any component can provide information about SCCs.*

5.3.2.2 Node-wise abstraction

Among the main algorithmic contributions of this thesis, the last one is a specialized abstraction that fits saturation and the presented incremental symbolic SCC detection algorithm, as well as it complements Proposition 2 (about recurring states) as a cheap way to prove the absence of SCCs. The goal of the following construct is to match the order in which events are processed during saturation, as well as the structure of the underlying decision diagram.

Definition 33 (Node-wise abstraction) Node-wise abstraction of state graph $(\mathcal{S}, \mathcal{N})$ with regard to node n is $\mathcal{A}_n^{\exists} = (\mathcal{S}_n, \mathcal{N}_n^{\exists})$, where $\mathcal{S}_n = \{i \mid S(n[i]) \neq \varnothing\}$, i. e. reachable local states encoded by the arcs of n, and $\mathcal{N}_n^{\exists} = \{(s_k, s_k') \mid s_k, s_k' \in \mathcal{S}_n, \exists((\dots, s_k, \dots), (\dots, s_k', \dots)) \in \mathcal{N}_k\}$, i. e. the projections of events $\mathcal{E}_k$ to component k. ▪

Node-wise abstraction can be regarded as a projection of the submodel $M_{\leq k}$ to component k. With this hybrid, variables above level k (where recursive saturation will not change anything) are discarded in a must abstraction fashion, while variables below level k (where recursive saturation may explore further) are omitted by a projection.

Just like may abstractions, a series of node-wise abstractions can also be used to reason about global SCCs. Moreover, this can be done inductively during the state space generation. The following theorem gives the basis for this inductive method of using node-wise abstractions to prove the absence of SCCs.

Theorem 3 (Node-wise abstraction and SCCs) Given a node-wise abstraction $\mathcal{A}_n^{\exists}$ with regard to a saturated node n, the state graph $(S(n), \mathcal{N}_{\leq k})$ does not contain any SCC if the following assumptions hold: *1)* neither the abstract state graph $\mathcal{A}_n^{\exists}$ *2)* nor the state spaces $(S_{[i]}(n), \mathcal{N}_{<k})$ belonging to the children of n contain an SCC. ▪

Proof *Indirect proof. Suppose that $(S(n), \mathcal{N}_{\leq k})$ contains an SCC with a state in $S_{[i]}(n)$ (i. e. component k is in local state i). There are three possible cases to realize this:*

1. *The SCC is fully within $S_{[i]}(n)$ and contains transitions only from $\mathcal{N}_{<k}$;*
2. *The SCC is fully within $S_{[i]}(n)$ and contains transitions from $\mathcal{N}_k$, but they do not change the local state of component k;*
3. *The SCC contains a state from at least one other $S_{[j]}(n)$ $(i \neq j)$ as well.*

Case 1 contradicts Assumption 2, while case 2 is also a contradiction with Assumption 1, since not changing the local state of component k would mean a self loop in $\mathcal{A}_n^{\exists}$ constituting a trivial SCC. Proving that case 3 also yields a contradiction is based on Lemma 3. To reach the state in $S_{[j]}(n)$ from the state in $S_{[i]}(n)$, a path has to use at least one transition from $\mathcal{N}_k$. These transitions will also form a path in $\mathcal{A}_n^{\exists}$ from i to j. To be in an SCC, there has to be a path from the state in $S_{[i]}(n)$ to the state in $S_{[j]}(n)$ also using at least one transition from $\mathcal{N}_k$, which in turn also forms a path in $\mathcal{A}_n^{\exists}$ from j to i. Since i and j are reachable from each other, they are in an SCC of $\mathcal{A}_n^{\exists}$, which is again a contradiction with Assumption 1.

The main idea of the proof is that node-wise abstraction represents the effects of the events $\mathcal{E}_k$ exactly on the level of their *Top* value. At the time a node n becomes saturated, the only transitions that can change the local state of component k are in $\mathcal{N}_k$. Node-wise abstractions contain the images of exactly these transitions, thus they describe the possible transitions between sets of partial states encoded by the children of n. This is why they can be used to identify so-called *one-way walls* in the state space that separate the possible spaces for SCCs.

Note that the theorem did not specify how to ensure Assumption 2 (SCC-free state spaces of children nodes). This means that the algorithm is not restricted to using node-wise abstraction to prove SCC-freeness for the children – even if the corresponding node-wise abstraction did contain an SCC (which only implies the *possible* presence of a global SCC), the symbolic fixed point computation algorithm of Section 5.2.2 can check if the abstract SCC candidate is realizable in the global state space or not. This way, the series of saturated nodes can inductively prove the absence of SCCs by the end of the state space generation.

Example 2 *Observing Figure 5.5 again, node-wise abstractions of the state space with regard to the three decision diagram nodes can be seen in Figure 5.5d. As Theorem 3 suggests, it is unnecessary to start symbolic SCC detection until the top level node n_3 (corresponding to the consumer component) gets saturated, since it is the only node whose node-wise abstraction contains an SCC.*

By the time a node is saturated, its node-wise abstraction will not change anymore. This way, a single abstraction has to be built and analyzed only once. The computation of node-wise abstractions is very simple and cheap. It can be done on demand by projecting the next-state relation of corresponding events to the Top component, or on-the-fly during saturation by adding vertices and arcs each time a new local state is discovered or a new transition of the corresponding events is fired, respectively (marked by a diamond in Figure 5.6). A simple must abstraction can also be examined as part of computing SCCs of the node-wise abstraction by checking if every local transition used in the SCC belongs to events having only the current component as a supporting one. If this is the case, that SCC is inherently realizable and can be returned as a counterexample immediately.

5.3.3 On-the-fly Incremental Hybrid Model Checking of LTL Properties

In this section, the building blocks presented so far are assembled into an on-the-fly, hybrid and incremental LTL model checking algorithm. Section 5.3.3.1 will show a way to reuse recurring states and the modified relational product operator in the symbolic fixed point computation algorithm. Section 5.3.3.2 shows how to consider recurring and accepting states in the explicit search, while Section 5.3.3.3 will integrate the symbolic and explicit searches. Section 5.3.4 will present the full hybrid SCC detection algorithm.

5.3.3.1 Recurring States in the Fixed Point Computation

In addition to indicating the absence of SCCs, recurring states can also help in finding them. Since each SccSaturate builds its own set of recurring states, the set of all gathered recurring states on level k will all be target states of transitions in $\mathcal{N}_k$ (only these transitions are fired on level k). This way, the "interesting" subset of $\mathcal{S}^+$ is available at the end of a Saturate call and DetectSCC can be initialized with the recurring states instead of $S(n)$.

Corollary 1 (Accelerating the fixed point computation) During an SCC detection phase after the saturation of node n, restricting the set of "new" transitions $\mathcal{N}_k$ to those that end in recurring states (that is, $\{(\mathbf{s}, \mathbf{s}') \mid (\mathbf{s}, \mathbf{s}') \in \mathcal{N}_k, \mathbf{s}' \in \mathcal{R}\}$) will still find all SCCs that would be found otherwise. ▪

The corollary follows from the fixed point computation strategy of saturation and its caching mechanisms. It can be used to accelerate the fixed point computation by using a smaller input set, since a smaller set of "new" transitions makes DetectSCC finish in fewer iterations.

Another application of RelProd is the computation of the filtered sets in the filtering loop of DetectSCC. Since RelProd can be used to compute the intersection of the relational product and any other set of states on the fly, it is suitable to substitute the intersection used in the computation of reachable states.

5.3.3.2 Constraining the Explicit Search of Abstractions

Recurring states together with accepting states (if any) are useful in the explicit search on node-wise abstractions as well. According to Proposition 2, every SCC has to contain at least one recurring state.

If fair SCCs are sought, then at least one state of an acceptance set $\mathcal{F}$ also has to be included in the SCC.

In a node-wise abstraction $\mathcal{A}_n^{\exists}$, every node i represents the set of states $S_{[i]}(n)$. If the SCC candidate is realizable, it will contain some states from these sets. A necessary property of abstract SCC candidates is therefore described by the following corollary that is a direct consequence of the above considerations.

Corollary 2 (Necessary condition for abstract SCC realizability) An abstract candidate SCC $\Theta_{\mathcal{A}} = (\mathcal{S}_\Theta, \mathcal{N}_\Theta)$ of a node-wise abstraction $\mathcal{A}_n^{\exists}$ is not realizable or not fair if

- $\left(\bigcup_{i \in \mathcal{S}_\Theta} S_{[i]}(n)\right) \cap \mathcal{R} = \emptyset$, or
- $\left(\bigcup_{i \in \mathcal{S}_\Theta} S_{[i]}(n)\right) \cap \mathcal{F} = \emptyset$, respectively. ▪

Since every $S_{[i]}(n)$ as well as $\mathcal{R}$, and even the set of accepting states (formally $\{(\mathbf{s}, q) \mid q \in \mathcal{F}\}$) is known by the time of the explicit search, candidate SCCs can be evaluated and sorted out if the necessary conditions do not hold. If no potentially realizable candidate SCCs remain, the state space does not contain any global SCCs either.

5.3.3.3 SCC Candidates in the Fixed Point Computation

Node-wise abstraction and the incremental symbolic fixed point computation algorithm are strongly related. Node-wise abstraction contains exactly those transitions that are considered "new" in the fixed point algorithm, so the latter can be regarded as a method to check whether a candidate SCC is realizable or not.

According to Definition 33 (the definition of node-wise abstraction) and Theorem 3, arcs of candidate SCCs represent the transitions that *may* be part of an SCC – if an arc is not part of any candidate SCC, its corresponding transitions will not be part of a global SCC either.

Corollary 3 (Realizing abstract SCCs) Given a node-wise abstraction $\mathcal{A}_n^{\exists}$ and an abstract SCC candidate $\Theta = (\mathcal{S}_\Theta, \mathcal{N}_\Theta)$, the only transitions of $\mathcal{N}_k$ that can be part of the global SCC are those corresponding to abstract arcs $\mathcal{N}_\Theta$, formally $\{(\mathbf{s}, \mathbf{s}') \mid (\mathbf{s}, \mathbf{s}') \subseteq \mathcal{N}_k, (s_k, s'_k) \in \mathcal{N}_\Theta\}$, where s_k and s'_k are the local states of component k in the global states $\mathbf{s}$ and $\mathbf{s}'$. ▪

The corollary can be exploited in the symbolic fixed point algorithm by considering only those transitions as "new" that are part of a candidate SCC instead of the full relation $\mathcal{N}_k$.

5.3.4 Assembling the Pieces – The Full SCC Detection Algorithm

After introducing the different aspects and components of the incremental hybrid SCC detection algorithm presented in this chapter, this section summarizes the algorithm as a whole. The input model is assumed to be a PTS. SCC detection relies only on a base algorithm employing the iteration strategy of saturation, including, but not limited to traditional saturation, constrained saturation or the product computation algorithm of Section 4.
Whenever a node n becomes saturated, the following steps have to be executed:

1. The set of encoded states $S_{[i]}(n)$ and transitions $\mathcal{N}_k$ are checked against emptiness – if any of them is empty, SCC detection reports "no SCC".
2. The set of collected recurring states $\mathcal{R}$ and – if applicable – accepting states $\mathcal{F}$ are checked against emptiness – if no recurring states were found, SCC detection reports "no SCC" (see Proposition 2).

3. An explicit SCC computation algorithm is run on the current node-wise abstraction $\mathcal{A}_n^{\exists}$ to obtain an SCC candidate – if no candidate is found, SCC detection reports "no SCC" (see Theorem 3).
4. Candidate SCCs are checked according to Corollary 2 – if none of them is appropriate, SCC detection reports "no SCC".
5. A fixed point computation is started with $\mathcal{S}^+ = \mathcal{R}$ and transitions corresponding to arcs of the candidate SCCs as "new" transitions – the result of the computation is returned as the result of this iteration (see Lemma 4, Theorem 2 and Corollaries 1 and 3).

As summarized in Figure 5.3, the algorithm terminates as soon as DetectSCC finds a nonempty fixed point, or if the root node is saturated. If a nonempty fixed point is found, its contents can be used to aid counterexample generation (e. g. saturation-based SCC computation algorithms of [ZC11]). Otherwise, there is no counterexample and the model M is valid in terms of the specification φ.

Figures 5.6 and 5.7 show the pseudocodes of the modified Saturate and RelProd functions.

SCC detection with saturation is implemented in functions SccSaturate and SccRelProd (Figures 5.6 and 5.7, respectively). They are very similar to Saturate and RelProd, but without a constraint. Differences are marked by an asterisk in case of lines related to recurring states, a diamond marks the on-the-fly construction of the node-wise abstraction, and a circle shows the line where symbolic SCC detection is called.

SccRelProd has two additional parameters: o is a node encoding the "old states" that the return value will be merged with (in order to compute their intersection) and r is an in-out parameter used to return this intersection (the recurring states). Computation of this intersection is integrated with the function SccRelProd. When the recursion ends with reaching $\mathbf{1}$, o is checked. If it is $\mathbf{1}$, then the found new state is in fact already in the set of states that will be merged with the result, so r is also set to $\mathbf{1}$ to include the current state in the set of recurring states.

In order to collect recurring states, a node r is created in every SccSaturate call. During the computation of the relational product, another temporary node is passed to every call of SccRelProd as an in-out parameter to collect the recurring states of the lower levels. When the function returns, this node is merged into the corresponding child of r_k. SccRelProd also collects recurring states this way. Once t is saturated, the algorithm checks r to see if there is any recurring state collected.

The abstraction is built on the fly every time a local transition processed in SccSaturate is found to be globally fireable, i. e. SccRelProd returns a nonempty node. If recurring states are found, transitions that yielded an SCC in $\mathcal{A}_{t_k}^{\exists}$ are computed[2] and stored in $\mathcal{N}_\Theta$.

Finally, DetectSCC is called with the set of recurring states $S(r)$, $\mathcal{N}_{\leq k}$ as the set of all processed transitions and $\mathcal{N}_\Theta$ as the set of new transitions.[3] If it returns true (i. e. an SCC is detected), the whole algorithm is terminated with a counterexample.

5.4 Evaluation

To demonstrate the efficiency of the presented new model checking algorithm (referred to as *HybMC*), models of the Model Checking Contest[4] (MCC) have been used to compare it to three competitive tools. NuSMV 2 [Cim+02] is a BDD-based model checker implementing traditional SCC-hull algorithms and is well-established in the industrial and academical community. Its successor, nuXmv

[2]This step can also be done on the fly inside DetectSCC.

[3]Checking the emptiness of $\mathcal{N}_\Theta$ is not necessary, because DetectSCC will terminate almost immediately if $\mathcal{N}_{new}$ is empty.

[4]http://mcc.lip6.fr/

```
    input   : $n$ : node
    // $n$: node to be saturated,
    output  : node
  1 if $n = \mathbf{1}$ then return $\mathbf{1}$
  2
  3 $m \leftarrow \mathcal{N}_k$ as MxD
  4 Return result from cache if possible
  5 $n' \leftarrow new\ Node_k$
  6 for each $i \in \mathcal{S}_k : n[i] \neq \mathbf{0}$ do $n'[i] \leftarrow$ SccSaturate$(n[i])$
  7
* 8 $r \leftarrow new\ Node_k$
  9 repeat
 10 |  foreach $n[i] \neq \mathbf{0} \wedge m[i][i'] \neq \mathbf{0}$ do
*11 |  |  $r'_{k-1} \leftarrow new\ Node_{k-1}$
 12 |  |  $u_{k-1} \leftarrow$ (SccRelProd$(n'[i], m[i][i'], n'[i'], r'_{k-1})$)
◇13 |  |  if $u_{k-1} \neq \mathbf{0}$ then add arc $(i, i')$ to $\mathcal{A}^{\exists}_{n'}$
 14 |  |  $n'[i'] \leftarrow (n'[i'] \cup u_{k-1})$
*15 |  |  $r[i'] \leftarrow (r[i'] \cup r'_{k-1})$
 16 |  end
 17 until $n'$ unchanged
*18 $r \leftarrow PutInUniqueTable(r)$
 19 $n' \leftarrow PutInUniqueTable(n')$
 20 Put inputs and results in cache
*21 if $r = \mathbf{0}$ then return $n'$
◇22 $\mathcal{N}_\Theta \leftarrow$ transitions corresponding to SCCs of $\mathcal{A}^{\exists}_{n'}$
○23 if DetectSCC$(S(r), \mathcal{N}_{\leq k}, \mathcal{N}_\Theta)$ then terminate with counterexample
 24
 25 return $n'$
```

Figure 5.6: SccSaturate

```
    input   : $n, m, o$ : node
    // $n$: node to be saturated,
    // $m$: next-state node,
    // $o$: old node,
    in-out  : $r$ : node
    // $r$: recurring states
    output  : node
  1 if $n = \mathbf{1} \wedge m = \mathbf{1}$ then
* 2 |  if $o = \mathbf{1}$ then $r \leftarrow \mathbf{1}$
  3 |  return $\mathbf{1}$
  4 end
  5 Return result from cache if possible
  6 $n' \leftarrow new\ Node_k$
  7 foreach $n[i] \neq \mathbf{0} \wedge m[i][i'] \neq \mathbf{0}$ do
* 8 |  $r'_{k-1} \leftarrow new\ Node_{k-1}$
  9 |  $n'[i'] \leftarrow (n'[i'] \cup$ SccRelProd$(n'[i], m[i][i'], n'[i'], r'_{k-1}))$
*10 |  $r[i'] \leftarrow (r[i'] \cup r'_{k-1})$
 11 end
*12 $r \leftarrow PutInUniqueTable(r)$
 13 $n' \leftarrow$ SccSaturate$(PutInUniqueTable(n'))$
 14 Put inputs and results in cache
 15 return $n'$
```

Figure 5.7: SccRelProd

[Cav+14] –among other methods– implements the IC3 algorithm for LTL model checking. ITS-LTL [Dur+11b] is a powerful tool based on saturation that implements various optimizations both for the symbolic encoding and on-the-fly SCC detection.

5.4.1 Implementation

The algorithms presented in Chapter 4 and this chapter were implemented in the PetriDotNet[5] framework. PetriDotNet is a modeling and model checking tool written in C# supporting colored and ordinary Petri nets. It has been developed by the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics, including the author of this dissertation. It supports basic analysis of Petri nets, and a handful of saturation-based model checking techniques including state space generation and reachability analysis, CTL model checking, bounded CTL model checking and on-the-fly LTL model checking (presented in this thesis). Here we only use it as an interface to models, and we reuse the basic saturation algorithms and related data structures from the previous CTL model checking algorithms. The implementation of our new method also uses tools of the SPOT toolset [DP04] to parse LTL formulas and transform them to Büchi automata.

5.4.2 Benchmark Cases

The Model Checking Contest offers Petri net models of various artificial and real-world problems. The models are given in PNML format [Hil+09], usually both as colored Petri nets and unfolded Petri nets as well. Due to the supported input formats of the selected tools, only ordinary Petri nets could be used.

Excluding the "surprise models" of the contest of 2014 (which were released after conducting the measurements), a total of 27 scalable models were used in the benchmark, with instances of different size, resulting in 157 model instances. The majority of the models expose concurrent and asynchronous behavior. Except the "planning" model, state spaces of the nets are finite. Even the infinite model was kept in order to demonstrate that on-the-fly LTL model checking can sometime bear with infinite models: if the product is finite or a counterexample is found in a finite subset of the state space.

As for the specifications, a tool of the SPOT toolset was used to generate real (i. e. no pure Boolean) LTL formulas with predefined atomic propositions. Atomic propositions were generated based on the models: for every place of the smallest instance (those that appear in instances of any size) two propositions requiring zero and nonzero token counts were defined. SPOT generated 50 properties for every model class – instances of every size were checked against these properties. All in all, the 50 properties for each of the 157 model instances gave a total of 7 850 benchmark cases.

5.4.2.1 Generated LTL formulas

There are many categorizations of LTL formulas based on syntactic or semantic considerations [MP92]. The generated LTL properties were also categorized by a tool of SPOT. The following categories were used in the measurements.[6]

- *Safety properties* (1 466 formulas): Specifies that something "bad" never happens. In general, counterexamples of these properties consist of a finite prefix that cannot be "fixed" with any suffix, i. e. the violation occurs in the prefix itself. In LTL, they are usually in the form G φ.

[5] http://petridotnet.inf.mit.bme.hu/en/

[6] The categories are not always exclusive. For example, obligation properties include safety and guarantee properties.

- *Guarantee properties* (2 112 formulas): Specifies that something “good” is guaranteed to happen. Counterexamples for guarantees are “lasso” shaped, since they have to describe an infinite behavior that fails to expose the desired property. In LTL, they are usually in the form $\mathsf{F}\ \varphi$.
- *Obligation properties* (4 975 formulas): Combination of safety and guarantee properties. In LTL, they are usually in the form $\mathsf{G}\ \varphi_1 \vee \mathsf{F}\ \varphi_2$.
- *Pure eventuality formulas* (1 620 formulas): If φ is a pure eventuality formula and the path ρ models φ, then $\zeta\rho$ also models φ, where $\zeta \in \mathcal{S}^*$ is any finite prefix. In other words, pure eventualities describe *left-append closed* languages.
- *Pure universality formulas* (1 260 formulas): If φ is a pure universality formula and the path $\zeta\rho$ models φ, then ρ also models φ, where $\zeta \in \mathcal{S}^*$ is any finite prefix. In other words, pure universalities describe *suffix closed* languages.

There are some more complex and interesting categories that cannot be automatically recognized by SPOT. For this reason, the category “not obligation” (2 875 formulas) is used for such properties, including the following categories.

- *Progress properties*: Specifies that something “good” will keep happening again and again. In LTL, they are usually in the form $\mathsf{G}\ \mathsf{F}\ \varphi$.
- *Response properties*: Specifies that a response will eventually be given to a certain event whenever it occurs. In LTL, they are usually in the form $\mathsf{G}\ \varphi_1 \Rightarrow \mathsf{F}\ \varphi_2$.
- *Stability properties*: Specifies that a certain property will stabilize eventually. In LTL, they are usually in the form $\mathsf{F}\ \mathsf{G}\ \varphi$.

As it turned out, the presented algorithm is not sensitive to the category of the checked LTL property.

5.4.2.2 Tools and Inputs

As mentioned, the tools selected for comparison are NuSMV, nuXmv and ITS-LTL.[7] Unfortunately, at the time of the measurements, PNML was not supported in any of the tools selected for comparison. Both NuSMV and nuXmv consume models in their native SMV format, while ITS supported many types of models and formats other than PNML. In case of NuSMV and nuXmv, PNML models were translated to SMV by our model exporter in PetriDotNet, similarly to the approach discussed in [SB14]. Since ITS accepts models in CAMI format (the native format of the tool CPN-AMI) and there exists a widely-used tool to convert between PNML and CAMI, this tool was used to generate the input files for ITS.[8]

In terms of the properties, both Hyb-MC and ITS-LTL uses SPOT to parse the formula and transform it into a Büchi automaton. A small tool was implemented to transform these formulas into the format of NuSMV and nuXmv.

[7]While there were many other powerful model checkers that could process LTL at the time of this research, we were interested in symbolic model checkers, preferably using a hybrid approach. Only ITS-LTL satisfied this requirement, but we included NuSMV as an established baseline among BDD-based model checkers as well as nuXmv that is a SAT/SMT-based symbolic model checker that represented the state of the art of its category at that time.

[8]The conversion between different input formats might have an impact on the performance of the tools. While ITS can handle Petri nets directly (only the file format has to be changed), the models have to be transformed into SMV format for the NuSMV and nuXmv tools. Direct, manual modeling could have yielded a more efficient model, but due to the number of benchmark models, automatic conversion had to be used.

5.4.2.3 Benchmark settings

Measurements were done on identical server machines with Intel Xeon processors (4 cores, 2.2GHz) and 8 GB of RAM. Hyb-MC and ITS-LTL were run on Windows 7 x64, while NuSMV and nuXmv were run on CentOS 6.5 x64.

The timeout of each measurement was set to 600 seconds. Runtimes were measured internally by every tool. In case of Hyb-MC, the runtime includes the processing time of SPOT (transformation to automaton) and the total runtime of the algorithm including its initialization, but not the loading time of PetriDotNet. ITS-LTL also measured its runtime internally, also including the runtime of SPOT and the algorithm. NuSMV and nuXmv can measure time by placing a special command in the input script. In case of these tools, only the actual runtime of model checking was measured, omitting the time of loading the model and constructing the binary model from the SMV input, making their measured runtimes slightly smaller than the actual runtime. In total, approximately 40 days of processing time was spent on the evaluation.

The decision diagram based tools (Hyb-MC, NuSMV and ITS-LTL) used the same variable ordering produced by heuristics of the ITS toolset. Out of the numerous Python scripts bundled with ITS-LTL, “Script 11” was used to produce a flat ordering with each place encoded in a separate variable – this produced orderings that every tool could parse and use. This way, the hierarchical features of ITS might not have been fully exploited, but saturation-based algorithms in PetriDotNet can also be faster if a variable can encode multiple places.

5.4.3 Results

The following sections present some aspects of the results of the evaluation. Overall, 5 megabytes of measurement log was collected and analyzed. Out of the successfully checked cases, properties were fulfilled 2 811 times, while 3 565 cases gave counterexamples. In 1 474 cases, all the tools exceeded the predefined time limit. The whole benchmark and the analyzed results can be downloaded from the website[9] of PetriDotNet dedicated to [c8].

5.4.3.1 Comparison of Runtime Results

The main emphasis of preliminary analysis was on the runtime results of each tool. The runtime of Hyb-MC was compared to the corresponding runtime of the other tools. In addition, simple state space generation with saturation was also done for each model instance to prove that saturation is not the single reason of high performance—on-the-fly model checking can finish much earlier than a simple state space generation.

Results can be seen on the scatter plots of Figure 5.8. The four scatter plots show the comparison of Hyb-MC to simple state space generation (SSG) with saturation and the three chosen tools. Each dot represents a benchmark case. The horizontal and vertical axes measure the runtime of Hyb-MC and the other tool, respectively. A dot above the diagonal line is a benchmark case that Hyb-MC solved faster than the competitor. The borders of the diagrams represent cases where one of the tools did not finish under the time limit. Cases in which neither of the tools finished are not shown on this diagram.

As the top-left plot suggests, model checking with Hyb-MC is usually faster than simple state space generation. The main cause of this is the algorithm’s on-the-fly operation and efficient incremental operation (i. e. a low degree of redundancy). There are also some cases where state space

[9] http://inf.mit.bme.hu/en/tacas15.

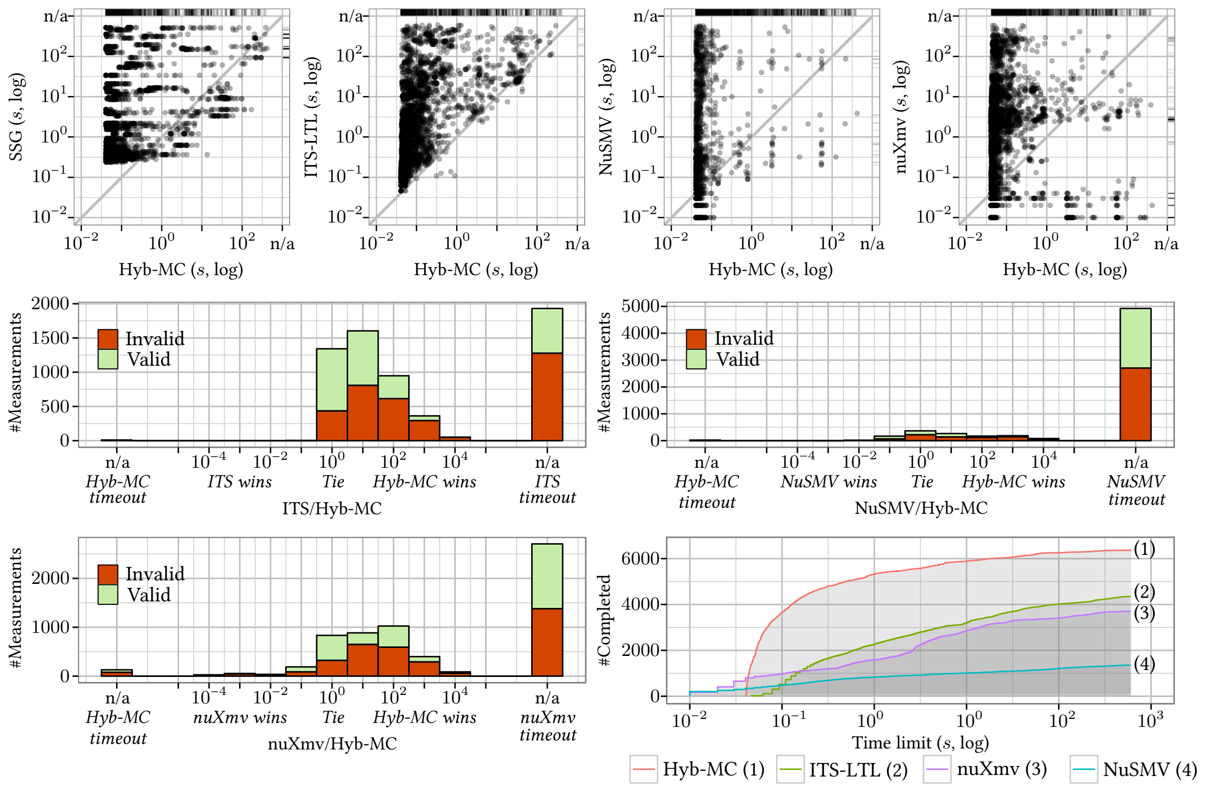


Figure 5.8: Measurement results.

generation could be finished, but model checking was unsuccessful. These cases show that incremental operation and the presented optimizations cannot always compensate the overhead of model checking complex properties.

Comparing to the other model checking tools, the vast majority of cases show the competitiveness of the presented algorithm. Since the scales of the axes are logarithmic, the distance of a dot from the diagonal indicate an exponential difference in the runtimes of the tools.

This difference is visualized on the bar charts of Figure 5.8. The bar charts show the quantity t_A/t_B with a logarithmic horizontal axis, where t_A and t_B are the runtime of the compared algorithms. Bar charts of Figure 5.8 are colored to show the distribution of valid and invalid properties. Hyb-MC is better than the other tools more often in refuting an invalid property than in proving valid ones. This shows the efficiency of the SCC detection algorithm compared to the other approaches, but even valid cases show a decent speedup compared to the other three tools, justifying the product computation algorithm and the use of abstraction and recurring states as a means to prove the absence of counterexamples.

The last diagram in Figure 5.8 shows how many of the cases finished with runtime under the value of the horizontal axis, i. e. how many cases would have finished if the timeout was set to a specific value. This diagram indirectly suggests the scalability of the algorithms. The initial delay of Hyb-MC and ITS-LTL is due to SPOT building the Büchi automaton from the LTL property.

It is interesting to note some special features of the tools. ITS-LTL performed the best among the chosen competitors, but it could outperform Hyb-MC in very few cases only. Since it is also based on saturation and uses abstraction to perform on-the-fly model checking, these results are the most

Table 5.1: SCC detection statistics in different groups of cases.

Cases	Time spent with SCC detection	Prevented symbolic runs	Prevented by Recurring	Prevented by Abstraction
All	16.9%	99.7%	89.7%	10.3%
Valid	13.5%	99.7%	89.0%	11.0%
Invalid	19.6%	99.7%	91.3%	8.7%
Obligation	13.7%	99.7%	86.3%	13.7%
Not obligation	23.2%	99.7%	93.0%	7.0%
Echo (2D)	2.7%	100.0%	100.0%	0.0%
Eratosthenes	65.9%	62.9%	19.0%	81.0%

significant among all the others.

Although NuSMV performed the worst of all competitors, it could beat Hyb-MC in more cases and also more significantly than ITS-LTL. NuSMV is the oldest of all the tools and is supposed to be better with synchronous models, so these cases deserve future attention.

Results of nuXmv are very different from results of the other tools. This is not surprising, since it uses a SAT-based algorithm with different techniques and strengths. It is also the one that outperformed Hyb-MC most of the times, it even managed to process many cases that caused a timeout in Hyb-MC. It is also interesting to note that nuXmv proved to be the only tool sensitive to the category of the checked property.

Analysis of runtime results measured on model families showed other interesting differences in the scalability of the tools. NuSMV and nuXmv performed significantly better on models where instances of the model were scaled in the domain of state variables. On the other hand, saturation-based tools Hyb-MC and ITS-LTL were much better on models that scaled in the number of variables (components).

5.4.3.2 Efficiency of the Presented Techniques

During the measurements, Hyb-MC spent only 17% of the time computing SCCs. Overall, 359 084 symbolic fixed point computations were started, while abstraction and explicit algorithms prevented $1.22 \cdot 10^8$ runs of symbolic SCC computation, 99.7% of all the cases. 90% of these cases were prevented by the absence of recurring states (as a first check), while the remaining 10% were the cases where explicit runs on node-wise abstractions managed to find even more evidence.

Table 5.1 shows the discussed statistics for some interesting subsets of the benchmark cases. Not surprisingly, Hyb-MC spends more time looking for SCCs in case of invalid properties. The efficiency of recurring states and explicit search slightly varies, but not significantly. The difference is greater between obligation and more complex formulas: complex properties (such as progress properties) require more time spent on SCC detection. The share of recurring states and explicit search also varies a bit more.

Echo (2-dimensional instances) and Eratosthenes are two extreme models in terms of time spent on SCC detection. Echo was one of the hardest models for Hyb-MC, only 26% of all the cases were solved in the 2-dimensional family, all of which were valid. The fact that symbolic SCC detection was never called in these cases suggests that Hyb-MC fails here on invalid cases, where SCC detection would be necessary (cases that timed out do not contribute to these statistics). Eratosthenes, on the other hand, had all of its cases solved, and there also were valid properties. In spite of this, almost two thirds of the time was spent on computing SCCs. Explicit search also shows extremely high efficiency here, although the percentage of prevented symbolic runs is the only value among the models that is under 90%.

5.5 Summary and Future Work

LTL model checking of asynchronous systems is a computationally difficult problem. Various techniques and algorithms were developed in the history to tackle the state space explosion problem which is inherent in concurrent systems, and to combat the complexity of LTL model checking. I address this complex problem in my work by introducing a new, hybrid LTL model checking algorithm for asynchronous systems. The proposed approach combines the advantages of various algorithms in a novel way. Saturation explores the possible states of the system and constructs the synchronous product on the fly with the help of a synchronization method built on top of constrained saturation.

A new incremental fixed-point algorithm decomposes the model checking problem into smaller, component-wise queries and computes local model checking results. In order to decrease the number of symbolic fixed point computations, the thesis introduces a scalable abstraction framework, which efficiently filters SCC computations by using local explicit model checking runs.

Thesis 2 includes the following new algorithms (in Chapters 4 and 5):

- An efficient on-the-fly synchronous product generation algorithm based on saturation;
- An incremental symbolic fixed point computation algorithm for SCC detection;
- An abstraction technique to support inductive reasoning on the absence of SCCs;
- A unique hybrid model checking algorithm combining explicit state traversal with symbolic state space representation.

As a theoretical result, the thesis also contains the proofs for the correctness of the presented algorithms.

The new algorithms were implemented in the PetriDotNet framework and extensive measurements were conducted to examine efficiency. The tool was compared to industrial and academic model checking tools. Although the implementation was only in the prototype phase, it turned out to be quite competitive and could solve more of the benchmarks cases than any of its competitors. In addition, according to Yann Thierry-Mieg, the current ITS-tools implementation (in 2019) now uses solutions inspired by the results of Chapter 5.

Thesis 2 I designed an incremental, on-the-fly algorithm for the model checking of properties described by linear temporal logic (LTL), extending the saturation algorithm for state space generation and using it for fixed point computations.

2.1 I extended the saturation algorithm to directly generate the state space of the product system obtained by combining the system-under-analysis and a Büchi automaton describing the LTL property. The advantage of direct generation is that it avoids the explicit computation of the product transition relation.

2.2 I designed an algorithm to incrementally search for strongly connected components (SCC) in the state space during its generation by the saturation algorithm, using the saturation algorithm to compute the necessary fixed points. This approach enables on-the-fly model checking, i. e. the algorithm can terminate as soon as a witness is found.

2.3 I introduced two heuristics that complement the incremental SCC detection algorithm. Using abstraction-based techniques and the concept of recurring states, the heuristics can prove the absence of an SCC and therefore can speed up the search by preventing unnecessary fixed point computations.

2.4 I evaluated the resulting LTL model checking algorithm on models of the Model Checking Contest (MCC), comparing the runtime with three tools that represented the state of the art: the algorithm was found to be often orders of magnitude faster than its competitors.

The presented algorithm has a huge potential for future developments. Following the idea of driving the symbolic algorithm with explicit runs, a promising direction is to combine partial order reduction with symbolic model checking. In addition, advanced representations of the properties can also be used to further improve the speed of model checking.

Chapter 6

Enhancement and Generalization of the Saturation Algorithm

6.1 Introduction

As we could see in the previous chapters, locality has a crucial effect on the efficiency of saturation. Constrained saturation introduced an idea that can be used to keep fixed point computations of saturation as low as possible. Building on constrained saturation, Chapter 3 proposed a new approach for the model checking of prioritized Petri nets and Chapter 4 introduced further extensions to support the verification of linear temporal logic (LTL), in particular the on-the-fly computation of the product of the system and the property automaton. Both of them proposed ways to preserve locality for a transition relation that is composed of simple transitions and additional constraints (such as enabledness based on priorities or synchronization between the system and the property automaton).

In this chapter, we propose a new algorithm for saturation that generalizes the attempts of preserving locality in the approaches above. We introduce *conditional locality* to relax the original notion of locality and automatically handle transition relations that previously required a form of constrained saturation to process efficiently (such as in Chapters 3 and 4). In addition to generalizing a family of algorithms, using conditional locality can increase the saturation effect, which is intuitively associated with better performance (suggested also by results of the previous theses). We investigate this effect in the context of Petri nets, where we empirically show that the *generalized saturation algorithm* (GSA) can be orders of magnitude faster than the original saturation algorithm (described in Section 2.4.2) and is virtually never slower.

The main motivation of conditional locality is to compute fixed points even more locally. Saturation ignores variables that are independent of the processed events to avoid computing the fixed point for each of their valuations. With conditional locality, we can ignore even those variables that are not written but read by an event (because they will not change), at the price of computing the fixed point as many times as the value of those variables would cause a different result. The intuition is that the resulting nodes will be part of the final decision diagram more often than those created by the original saturation algorithm, leading to less intermediate nodes and therefore improved performance.

The most important related work is [Min04], where the authors propose a method to *split* a transition relation such that the resulting relations are as local as possible. The key idea is to extract relations which do not depend on the variables higher in the variable ordering and therefore the method works well when the transition relation is a "sum" of such a relation and another one (i. e. $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$). Our approach also handles the cases when the relation is the result of "removing" certain cases from a transition that normally does not depend on a variable (i. e. $\mathcal{R} = \mathcal{R}_1 \setminus \mathcal{R}_2$). Another work that is similar in spirit is [Mei+14], where the dependencies of high-level transitions on state variables are more fine-grained than *dependent* and *independent*, which enables a more compact encoding and more efficient update of the transition relation. Our approach also refines this dependency relation to relax the notion of locality.

The key novelties introduced in this thesis are the following: *1)* the introduction of *conditional locality* to relax the original notion of locality; *2)* the *generalization* of a family of saturation-based algorithms using conditional locality ; and *3)* an *empirical demonstration* of the efficiency of the proposed approach on Petri nets. This chapter is structured as follows. Section 6.2 presents a generalizations of next-state relations that can be used with saturation. Section 6.3 introduces conditional locality and the generalized saturation algorithm. The empirical evaluation on Petri nets is in Section 6.5. Finally, Section 6.6 concludes the chapter.

6.2 Next-State Representations

Saturation has been designed with various next-state representations, most of them presented in Section 2.3.4. Pseudocodes in the previous chapters used some of them, but it is apparent that they do not change the main aspects of saturation. In the following definition, we characterize the minimum requirement towards a next-state representation to be "compatible" with saturation and, in particular, the enhanced version presented here. We will build on this notion heavily in the generalization of saturation variants.

Definition 34 (Abstract next-state diagram) An *abstract next-state diagram* (ANSD) over a set of variables V ($|V| = K$) and corresponding domains D is a tuple $(\mathcal{D}, lvl, next)$

- $\mathcal{D} = \sqcup_{i=0}^{K} \mathcal{D}_i$ is the set of *next-state descriptors* (NS descriptor or descriptor for short), where items of $\mathcal{D}_0$ are the terminal identity $\mathbf{1}$ and the terminal empty $\mathbf{0}$ descriptors, the rest ($\mathcal{D}_{>0} = \mathcal{D} \setminus \mathcal{D}_0$) are *non-terminal* descriptors;
- $lvl : \mathcal{D} \to \{0, 1, \dots, K\}$ assigns non-negative *level numbers* to each descriptor, associating them with variables (descriptors in $\mathcal{D}_k = \{d \in \mathcal{D} \mid lvl(d) = k\}$ belong to variable x_k for $1 \le k \le K$ and are terminal nodes for $k = 0$);
- $next : \mathcal{D} \times \mathbb{N} \times \mathbb{N} \to \mathcal{D}$ is the indexing function that given a descriptor d and a pair of "before" and "after" variable values returns another descriptor d' such that $lvl(d') = lvl(d) - 1$ or $d' = \mathbf{0}$. Also denoted by $d[v, v'] = d' \Leftrightarrow (d, v, v', d') \in next$ (with $d, d' \in \mathcal{D}$, $v, v' \in \mathbb{N}$, $d[v, v']$ is left-associative) and $d[\mathbf{s}, \mathbf{s}'] = d[v_K, v'_K] \cdots [v_1, v'_1]$. We require for any $v, v', v'' \in \mathbb{N}$ and $v \neq v'$ that $\mathbf{1}[v, v] = \mathbf{1}$, $\mathbf{1}[v, v'] = \mathbf{0}$, and $\mathbf{0}[v, v''] = \mathbf{0}$. ∎

The abstract NS descriptor $d \in \mathcal{D}_k$ encodes the relation $\mathcal{N}(d) \subseteq \mathbb{N}^K \times \mathbb{N}^K$ iff for all $\mathbf{s}, \mathbf{s}' \in \mathbb{N}^K$ the following holds:

$$\big((\mathbf{s}, \mathbf{s}') \in \mathcal{N}(d) \Leftrightarrow d[\mathbf{s}, \mathbf{s}'] = \mathbf{1}\big) \wedge \big((\mathbf{s}, \mathbf{s}') \notin \mathcal{N}(d) \Leftrightarrow d[\mathbf{s}, \mathbf{s}'] = \mathbf{0}\big)$$

Decision diagram-based representations such as 2K-MDDs or matrix decision diagrams naturally implement ANSDs – descriptors are nodes of the diagram, the identity descriptor is the terminal one node ($\mathbf{1}$), the empty descriptor is the terminal zero node ($\mathbf{0}$) and the indexing is the same (in case of 2K-MDDs $d[x, x']$ is implemented by $d[x][x']$). The main difference between these representations and ANSDs is that the latter are *abstract* – they can have any representation as long as it can be mapped to the definition and they can be compared for equality.

Specialization for Petri Nets. In case of Petri nets, the simplest representation is the weight function of the net, which has been introduced as an implicit relation forest in Section 3.3. Another formalization of the same concept is given below, but this time as an ANSD. Given a Petri net with $K = |P|$ places each constituting a separate state variable (p_k denoting the kth variable in the ordering encoding the number of tokens on place $p \in P$), a mapping to an ANSD for every transition $t \in T$ is as follows.

- The set of descriptors is $\mathcal{D}_i = \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathcal{D}_{i-1}$ for $1 \leq i \leq K$, i. e. tuples of the input weight, inhibitor weight and output weight for p_i and the descriptor of for the next place if the transition is enabled with respect to p_i ($\mathcal{D}_0 = \{\mathbf{1}, \mathbf{0}\}$).
- For the *next* function, if $d = (v^-, v^\circ, v^+, d')$, the result of indexing is $d[i, j]$ is d' if $v^- \leq i < v^\circ$ and $j = i - v^- + v^+$ and $\mathbf{0}$ otherwise.
- For each transition t the corresponding descriptor is defined recursively: $d_0 = \mathbf{1}$ and $d_i = \big(W^-(p_i, t), W^\circ(p_i, t), W^+(p_i, t), d_{i-1}\big)$.

Relation to Homomorphisms. Abstract Next-State Diagrams can be related with homomorphisms over decision diagrams as used in ITS-tools [HTK08]. The key difference is that while an ANSD describes only a next-state relation and leaves the strategy of computing next-states to the state sace generation algorithm, homomorphisms describe the computation of obtaining the next-states for a set of states. This property lets model frontends of ITS-tools to customize the state space generation algorithm, and saturation itself is also implemented as a homomorphism. On the one hand, this provides great flexibility, but on the other hand it makes some of the optimizations proposed in this and the other two theses harder to implement in a general way. Constrained saturation, for example, is less optimal with homomorphisms in the general case (where the constraint is an arbitrary set of states and not predicates), because of an excessive loss of locality.

6.3 The Generalized Saturation Algorithm

As we could see, the motivation of the constrained saturation algorithm (and all of its variants presented in Theses 1 and 2) is to handle a modified transition relation without losing locality. This work generalizes these attempts by introducing the notion of conditional locality, a concept that expresses the most important consideration of all kinds of saturation: computing fixed points as locally (i. e. low in the decision diagram) as possible. This intuition has been discussed in Section 2.4, and the conclusion – that saturated nodes have a chance of being in the final MDD – can be used to improve the definitions to enhance this effect even further, which we do in the *generalized saturation algorithm* (GSA).

6.3.1 Conditional Locality

The concept of locality enables the saturation algorithm to ignore the value of variables outside the support of the currently processed event because it does not depend on them in any way. The result is that a fixed point can be calculated over partial states, which has to be computed *only once* regardless of the number of matching concrete states. The main motivation of conditional locality is to ignore even those variables that are not written but read by an event and compute the fixed point over even shorter partial states, but as many times as the value of those variables would cause a different result. The intuition is that the resulting nodes will be part of the final MDD more often than those created by the original saturation algorithm, leading to less intermediate nodes and therefore improved performance.

Definition 35 (Conditional locality) An event $e \in \mathcal{E}$ is said to be *conditionally local* over variables X and with respect to *condition variables* Y ($X \cap Y = \emptyset$) iff it is *local* over $X \cup Y$ and locally read-only on variables in Y. If Y is maximal and $X \cup Y = Supp(e)$, then we call $Y = Guard(e)$ the *guard variables* and $X = Supp_c(e)$ the *conditional support* of e. The variable with the highest index among the conditionally supporting variables (according to a variable order) is the *conditional top* variable ($Top_c(e)$) of e. ▪

The (full) next-state relation of a PTS can be automatically repartitioned based on conditional locality. The resulting partitions (events) will either be locally read-only on a variable or will always change its value (behaviors like "test-and-set" may combine these and be read-only sometimes but change the value other times – in this case, we can split the next-state relation). A special case of this repartitioning is built into the GSA as described in Section 6.3.2.

With conditional locality, the notion of submodel (over a set of variables X used in saturation is relaxed such that states are now not projected ot X, and all events e are included where $Supp_c(e) \subseteq X$. Just like a submodel, this *relaxed submodel* also includes a set of initial states, but they are not partial states now and the value they assign to $Y = V \setminus X$ is significant in defining the relaxed submodel (this will have implications on caching). What a relaxed submodel still promises is that variables in Y will not be changed by transitions of the relaxed submodel. Figure 6.1 illustrates the difference between a submodel and a relaxed submodel.

Definition 36 (Relaxed submodel of a PTS) Given a PTS $M = (V, D, \mathcal{I}, \mathcal{E}, \mathcal{N})$ and a set of target variables $V' \subset V$, another PTS $M' = (V', D, \mathcal{I}', \mathcal{E}', \mathcal{N}')$ is a *relaxed submodel* of M (denoted by $M \rightarrow_{V'} M'$) if:

- $\mathcal{I}' = \mathcal{I}$, i. e. initial states are the same;
- $\mathcal{E}' = \{e \mid Supp_c(e) \subseteq V'\}$, i. e. events are restricted to those of which are conditionally local over the set of target variables;
- $\mathcal{N}' = \bigcup_{e \in \mathcal{E}'} \mathcal{N}_e$, i. e. only conditionally local transitions are kept. ▪

The following definition of conditionally saturated sets of states and MDD nodes can be considered as relaxations of Definitions 23 and 24 based on conditional locality.

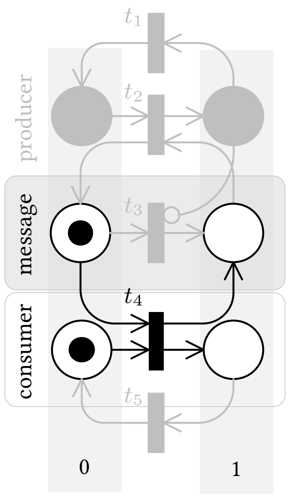
Definition 37 (Conditionally saturated set of states) Given a partitioned transition system M, a set of variables V' and the relaxed submodel $M \rightarrow_{x'} M'$ implied by the variables, a set of (partial) states S over variables X is conditionally saturated with respect to the partial state $\mathbf{s}_{(Y)}$ over variables $Y \subseteq V \setminus X$ iff $S' = S' \cup \mathcal{N}'(S')$, where $S' = \{\mathbf{s}_{(Y)}\} \times S_{(X)}$. ▪

Note that a set of (partial) states S over variables X that is conditionally saturated with respect to a zero-length state $\mathbf{s}_{(\emptyset)}$ is saturated over X, therefore the goal is the same as before: generate a minimal, conditionally saturated set of states S with respect to $\mathbf{s}_{(\emptyset)}$ that contains the initial states $\mathcal{I}$.

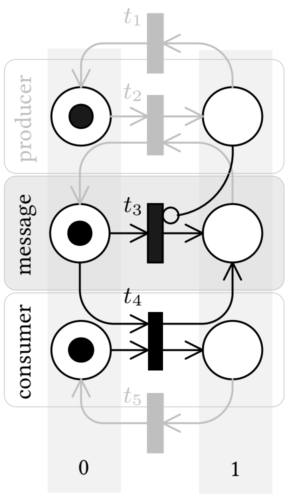
Definition 38 (Conditionally saturated node) Given a partitioned transition system M, an MDD node n on level $lvl(n) = k$ is *conditionally saturated* with respect to the partial state $\mathbf{s}_{(V_{>k})}$ over $V_{>k}$ iff it encodes a set of (partial) states $S(n)$ that is conditionally saturated with respect to the submodel corresponding to $V \leq k$ and $\mathbf{s}_{(V_{>k})}$. ▪

The equivalent definition in terms of child nodes is now phrased as a theorem.

Theorem 4 (Conditionally saturated node – recursive definition) Given a partitioned transition system M, an MDD node n on level $lvl(n) = k$ is conditionally saturated with respect to the



(a) As seen by saturation.



(b) As seen by the GSA.

Figure 6.1: The (relaxed) submodel of the example with inhibitor arc (Figure 2.1b) corresponding to the *message* component as seen by the original saturation algorithm as well as by the GSA. Notice that the GSA can decide the enabledness of t_3 but it will not change the state of the producer.

partial state $\mathbf{s}_{(V_{>k})}$ iff *1)* all of its children $n[i]$ are conditionally saturated with respect to $\mathbf{s}_{(V_{>k-1})}$, where $\mathbf{s}_{(V_{>k})} \searrow_{V>k} \mathbf{s}_{(V_{>k-1})}$ and $\mathbf{s}_{(V_{>k-1})}[k] = i$; and *2)* $S' = S' \cup \mathcal{N}_k(S')$, where $S' = \{\mathbf{s}_{(V_{>k})}\} \times S(n)$ and $\mathcal{N}_k = \bigcup_{e|Top_c(e)=x_k} \mathcal{N}_e$ for $1 \le k \le K$ and $\mathcal{N}_0 = \emptyset$. ▪

Proof *We prove only that a node described in the theorem encodes a conditionally saturated set of states. To prove the fixed point, we have to show that for any state* $\mathbf{s} \in \{\mathbf{s}_{(V_{>k})}\} \times S(n)$ *we have* $\mathcal{N}_{\le k}(\mathbf{s}) \subseteq \{\mathbf{s}_{(V_{>k})}\} \times S(n)$*. Note that* $\mathcal{N}_{\le k} = \bigcup_{i=0}^{k} \mathcal{N}_k$*. Assume there is a state* $\mathbf{s}' \in \mathcal{N}_{\le k}(\mathbf{s})$ *that is not in* $\{\mathbf{s}_{(V_{>k})}\} \times S(n)$*. We know that* $(\mathbf{s}, \mathbf{s}') \in \mathcal{N}_l$ *for some* $l \le k$*. If* $l = k$ *then we have a direct contradiction with the second requirement of the theorem. If* $l < k$*, then* $\mathbf{s}'[k] = \mathbf{s}[k] = i$*, because the transition cannot change the value of* x_k*. Because the first requirement of the theorem says that* $n[i]$ *is conditionally saturated with respect to* $\mathbf{s}_{(V_{>k-1})}$ *as defined above, and* $\mathcal{N}_l \subseteq \mathcal{N}_{\le k-1}$*, it follows that* $\mathbf{s}'$ *must be in* $\{\mathbf{s}_{(V_{>k-1})}\} \times S(n[i]) \subseteq \{\mathbf{s}_{(V_{>k})}\} \times S(n)$*.*

Based on Theorem 4 and the observation after Definition 37, the set of reachable states is encoded as a conditionally saturated MDD node on level K.

The key difference compared to Definitions 23 and 24 is the inclusion of a partial state with respect to which we can define a fixed point. Because we consider the repartitioned events that are now conditionally local, the partial state can be used to bind their guard variables, which will specify their effect on the variables in their conditional support. Since the guard variables are not changed when executing the transitions, we can compute a fixed point on only those variables that are in the conditional support of the event.

Even though the definition uses a partial state to define the fixed point, it is generally enough to traverse the NS descriptors just like the constraint in constrained saturation: whenever we navigate to $n[i]$, we should also navigate through $d[i, i]$. The resulting descriptor will characterize all the partial states that cause the same behavior in the rest of the transitions.

6.3.2 Detailed Description of the GSA

The pseudocode for the GSA is presented in Figure 6.2. The inputs are an MDD node n encoding the initial states $\mathcal{I}$ of a PTS, and a NS descriptor d representing the whole next-state relation $\mathcal{N}$. Since the algorithm will automatically partition the next-state relation based on conditional locality, d can be an union of all d_e (descriptors for events).

Sometimes, computing the full next-state relation is not practical, either because of its cost (e. g. we have to change representation) or because we want to use chaining in the fixed point computation. An advantage of ANSDs is the ability to represent operations in a lazy manner. For example, the union of two descriptors may be represented by extending the set of descriptors $\mathcal{D}$ with elements of $\mathcal{D} \times \mathcal{D} \times \{union\}$ ($\mathit{lvl}((d_1, d_2, union)) = \mathit{lvl}(d_1) = \mathit{lvl}(d_2)$) and extending *next* such that $(d_1, d_2, union)[i, j]$ is: $\mathbf{1}$ if d_1 or d_2 is $\mathbf{1}$; d_1 if d_2 is $\mathbf{0}$; d_2 if d_1 is $\mathbf{0}$; and $(d_1[i, j], d_2[i, j], union)$ otherwise. The *lazy descriptor* $(d_1, d_2, union)$ will not be equivalent to any non-lazy descriptor (even if they encode the same relation), but will be equivalent to $(d_1, d_2, union)$ or $(d_2, d_1, union)$, which is not optimal cache-wise but is often better than pre-computing the union. This approach can be generalized to more than two operands.

Compared to (constrained) saturation in Figures 2.7a–2.7c, the main differences and points of interest are listed below. In Saturate:

- Next-state descriptors are not retrieved for each level, but are a parameter.
- Recursive saturation of child nodes in line 5 passes $d[i, i]$ as the NS descriptor to use on the lower level $k - 1$, which encodes a set of transitions that do not modify the variable associated to this level (and any above), therefore they are conditionally local over $V_{\leq k-1}$ with respect to the partial state specified by the Saturate procedures currently on the call stack.
- Cache lookup in line 2 considers d instead of the partial state specified by the call stack because every partial state leading to d would produce the same result.
- In the fixed-point iteration in line 9 the Split procedure is used to retrieve the operands of a lazy union descriptor to support chaining. It may be implemented in any other way as long as the returned set of descriptors cover the relation encoded by the descriptor passed as argument.
- In lines 4 and 8, the Update procedure supports on-the-fly next-state relation building by providing a hook for replacing parts of d.

In SatFire:

- There are two descriptors: d_s for recursive saturation and d_f to fire.
- In the loop computing local successors in line 4 we omit locally read-only transitions ($i \neq j$), because they will be processed by recursive saturation.
- In the recursive firing in line 5, d_s is indexed by $[y, y]$ because (like in constrained saturation) the resulting node will be $n'[y]$ (and therefore $d_s[y, y]$ describes the conditionally local transitions), while d_f is indexed as usual.

In SatRecFire:

- Cache lookup in line 3 considers both next-state descriptors.
- In the loop computing local successors in line 5 we now *consider* every transition even if they are read-only, (on some level above they changed a variable).
- Recursive saturation in line 10 will use d_s (which is still conditionally local).

6.3.3 Discussion

To estimate the efficiency of the algorithm, we will consider the advantages and disadvantages of the different modifications. First and foremost it is important to note that if $\mathit{Top}_c(e) = \mathit{Top}(e)$ for every

```
Input: MDD node $n$, NS descriptor $d$
Output: saturated MDD node $n'$
if $n \in \{\mathbf{0},\mathbf{1}\}$ or $d \in \{\mathbf{0},\mathbf{1}\}$ then return $n$
if $\neg$SatCacheGet$(n, d, n')$ then  $n' \leftarrow$ new MddNode$(lvl(n))$
for each $i$ where $n[i] \neq \mathbf{0}$ do
    Confirm$(lvl(n), i)$, Update$(d)$
    $n'[i] \leftarrow$ Saturate$(n[i], d[i,i])$
end
repeat
    changed $\leftarrow$ false, Update$(d)$
    for each $d_f \in Split(d)$ do
        $n'' \leftarrow$ SatFire$(n', d, d_f)$
        if $n' \neq n''$ then  $n' \leftarrow n''$, changed $\leftarrow$ true
    end
until $\neg$changed
SatCachePut$(n, d, n')$

return $n'$
```

(a) Procedure Saturate.

```
Input: MDD node $n$, NS descriptor for saturate $d_s$, NS descriptor for fire $d_f$
Output: the result of firing $d$ from the states $n$ with the children saturated
if $n = \mathbf{0}$ or $d = \mathbf{0}$ then return $\mathbf{0}$
if $d = \mathbf{1}$ then return $n$
$n' \leftarrow$ new MddNode$(lvl(n))$
for each $x, y$ where $x \neq y$ and $d[x,y] \neq \mathbf{0}$ and $n[x] \neq \mathbf{0}$  do
    $m \leftarrow$ SatRecFire$(d[y,y], d[x,y], n[x])$
    if $m \neq \mathbf{0}$ then Confirm$(lvl(n), y)$
    $n'[y] \leftarrow n'[y] \cup m$
end
CheckIn$(n')$
return $n'$
```

(b) Procedure SatFire.

```
Input: NS descriptor for saturate $d_s$, NS descriptor for fire $d_f$, MDD node $n$
Output: saturated MDD node $n''$ (after firing $d_f$ from $n$ saturated with $d_s$)
if $n = \mathbf{0}$ or $d = \mathbf{0}$ then return $\mathbf{0}$
if $d = \mathbf{1}$ then return $n$
if $\neg$RecFireCacheGet$(d_s, d_f, n, n'')$ then
    $n' \leftarrow$ new MddNode$(lvl(n))$
    for each $x, y$ where $d[x,y] \neq \mathbf{0}$ and $n[x] \neq \mathbf{0}$  do
        $m \leftarrow$ SatRecFire$(d_s[y,y], d_f[x,y], n[x])$
        if $m \neq \mathbf{0}$ then Confirm$(lvl(n), y)$
        $n'[y] \leftarrow n'[y] \cup m$
    end
    CheckIn$(n')$, $n'' \leftarrow$ Saturate$(n', d_s)$, RecFireCachePut$(d_s, d_f, n, n'')$
end
return $n''$
```

(c) Procedure SatRecFire.

Figure 6.2: Pseudocode of the GSA.

event e, then the GSA degrades to the original saturation algorithm from [CMS06] or the corresponding constrained saturation algorithm from [ZC09], Chapter 3 or Chapter 4 with no difference in the iteration strategy and the virtually zero overhead of handling the next-state relation as a parameter. In every other case, there may be a complex interplay between the advantages and disadvantages discussed below.

An advantage of using conditional locality is that $\mathit{Top}_c(e) \leq \mathit{Top}(e)$, i. e. we can potentially use event e when saturating a node on a lower level, which is intuitively better because it raises the chance that the resulting node will be part of the final diagram. A direct price of this is the diversification of cache entries. By repartitioning the events, we may introduce a lot more next-state relations to process, and it is not evident if their smaller size and the enhanced saturation effect can compensate this. Furthermore, by keeping track of d_s (the descriptor to saturate with), we spoil the cache of saturation due to the following.

Whenever we navigate through $d[i,i]$, we remember something from i in the context of the next-state relation, yielding a potentially large number of different descriptors to saturate with. The original saturation algorithm saturates each MDD node only once, because it uses the same next-state relation every time. In the GSA, we saturate every pair of different MDD node and NS descriptor, so the diversity of descriptors can be a crucial factor. In the extreme case, when at least one event remembers every value along the path (for example because it copies them to other variables below), caching can degrade to the point where everything will need to be computed from scratch.

The other extreme is when each event remembers only one thing from the values bound above: whether it is enabled or not (e. g. when variables are compared to constants in guard expressions). Fortunately, this is the case with Petri nets: each transition will check variables locally and decide whether it is still enabled or not. This means that given a descriptor d representing transitions in $|T|$, the number of possible successors for $d[i,i]$ will be $O(|T|)$ (n values can partition $\mathbb{N}$ into $n+1$ partitions, each transition may contribute 2 values – one for an input arc and one for an inhibitor arc), but this number will also be limited by the number of non-zero child nodes of the saturated MDD node.

Given the facts that transitions of Petri nets are inherently conditionally local without repartitioning, and many nets are bounded (often safe), model checking of Petri net models with the GSA can be expected to yield favorable results. In fact, the experiment presented in Section 6.5 shows that the GSA is superior to the original saturation algorithm on every model that we analyzed.

For other types of models, we have yet to investigate the efficiency of the algorithm and the balance of benefits and overhead. It might be the case that we have to refine the read-only dependency into "local" and "global" evaluation (depending on whether we have to remember the value of the variable or can evaluate it immediately) and use conditional locality only with the "local" case. We also have to note that the efficient update of the next-state descriptors is not trivial and subject to future work.

6.4 Constrained Saturation Variants as Instances of the GSA

With the automatic partitioning offered by the GSA, next-state relations that motivated the introduction of constrained saturation and its variants can now be directly encoded into the transition relation without any cost. This is because a constraint is a *guard*, therefore it can cause an event only to become read-only on a variable instead of independent, but will still never write it. Adding a constraint will never raise the conditional top variable of events, but it can raise their unconditional top variable in many cases, which is associated with degraded performance.

Indeed, the handling of d_s in the GSA is very similar to the handling of the constraint node – we could say that our algorithm uses the next-state relation itself as a constraint. Combining this with the flexibility of abstract NS descriptors (lazy descriptors in particular), we get the properties of constrained saturation enhanced with every difference between the original saturation algorithm and the GSA (see Section 6.3.3).

We illustrate the usage of abstract NS descriptors for variants of constrained saturation with the kind of constraint used in the original constrained saturation algorithm [ZC09].

> **Definition 39 (Constrained next-state descriptor)** Given a NS descriptor d and a constraint node c, the *constrained next-state descriptor* d_c describing $\mathcal{N}(d_c) = \mathcal{N}(d) \setminus \big(\mathbb{N}^K \times S(c)\big)$ is a tuple $d_c = (d, c)$ with *lvl*(d_c) = *lvl*(d) = *lvl*(c), and $d_c[i,j]$ is: $\mathbf{0}$ if $d[i,j] = \mathbf{0}$ or $c[j] = \mathbf{0}$; and $(d[i,j], c[j])$ otherwise. ▪

For prioritized models, a similar definition can be used to remove transition which are not firable due to another enabled transition with higher priority. The EVIDD handle $h = \langle v, n \rangle$ encoding the minimal priority to fire will replace the constraint node c. The descriptor will be a triple instead of a pair, also containing the priority π assigned to transitions in d. When indexing the descriptor, $\pi \geq v$ will be checked instead of $c[j] = \mathbf{0}$ in the constrained version to see if d is fireable or not.

For the computation of the synchronous product, handling the constraint node c describing the transition relation of the automaton will be done inside the descriptor. When indexing a descriptor above level ℓ (the highest level on which the property automaton is encoded), StepConstraint will be used to compute $c[j]$. On and below level ℓ, $d[i,j]$ will be simply $c[i][j]$.

In case of LTL model checking, conditional locality will have additional benefits for SCC detection as well. Because events are processed on lower levels, counterexamples may be found sooner and in smaller submodels. As an extreme case, trivial fair SCCs (a self loop on an accepting state) will be detected on the terminal level, because it is conditionally local on no variable (i. e. it does not change any variable).

For the GSA, it is not important anymore how the next-state relation is encoded. Nevertheless, the lazy computation of these complex next-state relations are still powerful both with regard to memory consumption (the decomposed relations will usually be more compact, partly because simpler representations such as implicit relation forests can be used) and preprocessing time (sometimes an eager computation is not even possible because one of the relations is infinite). The main advantage of the GSA and ANSDs is exactly this: *optimizations on the next-state representation are now orthogonal to the saturation algorithm itself.*

6.5 Evaluation

In this section, we present the results of our experiments performed on a large set of Petri net models.

6.5.1 Research Questions

We have two main research questions about the GSA, both comparing it to the original saturation algorithm (SA) from [CMS06] (results should apply to constrained saturation as well). Both questions will be answered by measuring the relevant metrics for each algorithm and comparing the results for each benchmark model.

We expect that *1)* the GSA will be identical to the SA when conditional locality cannot be exploited; as well as in other cases *2)* the GSA will create less MDD nodes than the SA and *3)* in these cases it will be faster than the SA.

6.5.2 The Benchmark

We have implemented both the original saturation algorithm and the GSA in Java. Both variants used the same libraries for MDDs and next-state descriptors, and their source code differs only in the points discussed in Section 6.3.2.

We used the latest set of 743 available models from the Model Checking Contest 2018 [Kor+18], excluding only the Glycolytic and Pentose Phosphate Pathways (G-PPP) model with a parameter of 10–1000000000 (because the initial marking cannot be represented on 32-bit signed integers). We generated a variable ordering for each model using the *sloan* algorithm recommended by [Amp+18], and a *modified sloan* algorithm where we omitted read-only dependencies when building the dependency graph (motivated by the notion of conditional locality). We ran state space exploration 3 times on each model with each ordering, measuring several metrics of the algorithms. We will report the median of the running time of the algorithms (excluding the time of loading the model) and the total number of MDD nodes created during each run, as well as the size of the state space and the final MDD for each model and each ordering.

Measurements were conducted on a bare-metal server machine rented from the Oracle Cloud (BM.Standard.E2.64), with 64 cores and 512 GB of RAM, running Ubuntu 18.04 and Java 11. Three processes were run simultaneously, each with a maximum Java heap size of 100 GB and stack size of 512 MB. No process has run out of memory and the combined CPU utilization never exceeded 70%. Timeout was 20 minutes (including loading the model and writing results).

6.5.3 Results

The main results of the experiments can be seen in Figure 6.3. Every point represents a model (dashes on the side means a timeout), classified into two groups: "simply local" if none of the events had a read-only top variable and "conditionally local" otherwise. In the "simply local" group we expected no difference because the GSA should degrade into the original saturation algorithm, which was supported by the results. In the other group we were optimistic about the balance of advantages and disadvantages as discussed in Section 6.3.3, but the results were even better than what we expected. As the plots show, a significant part of the "conditionally local" models are below the reference diagonal, meaning that the GSA were often orders of magnitudes faster.

With the sloan ordering, 274 models were in the "conditionally local" group and the GSA was at least twice as fast as the SA in 53 cases. With the modified ordering, these numbers are 69 out of 298. In one case (*SmallOperatingSystem-MT0256DC0128*), the SA managed to finish just in time while the GSA exceeded the timeout (scaling was similar for smaller instances). Models where the GSA finished successfully but the SA exceeded the timeout with the sloan ordering include instances of *CloudDeployment*, *DiscoveryGPU*, *DLCround*, *DLCshifumi*, *EGFr*, *Eratosthenes*, *MAPKbis*, *Peterson* and *Raft*; and with the modified ordering also *AirplaneLD*, *BART*, *Dekker*, *FlexibleBarrier*, *NeoElection*, *ParamProductionCell*, *Philosopher*, *Ring* and *SharedMemory*. Analyzing these models in detail may provide insights about when the GSA is especially efficient.

Looking at the plots about the number of created MDD nodes (i. e. the size of the unique table) reveals that our expectations about less intermediate diagrams were correct and this probably has direct influence on the execution time. Even though not visible in Figure 6.3, interactive data analysis revealed that the model instances are more-or-less located at the same point on the execution time and node count plots. The collected data also suggests a linear relationship between the number of created nodes and the execution time, but this is rather a lower bound than a general prediction.

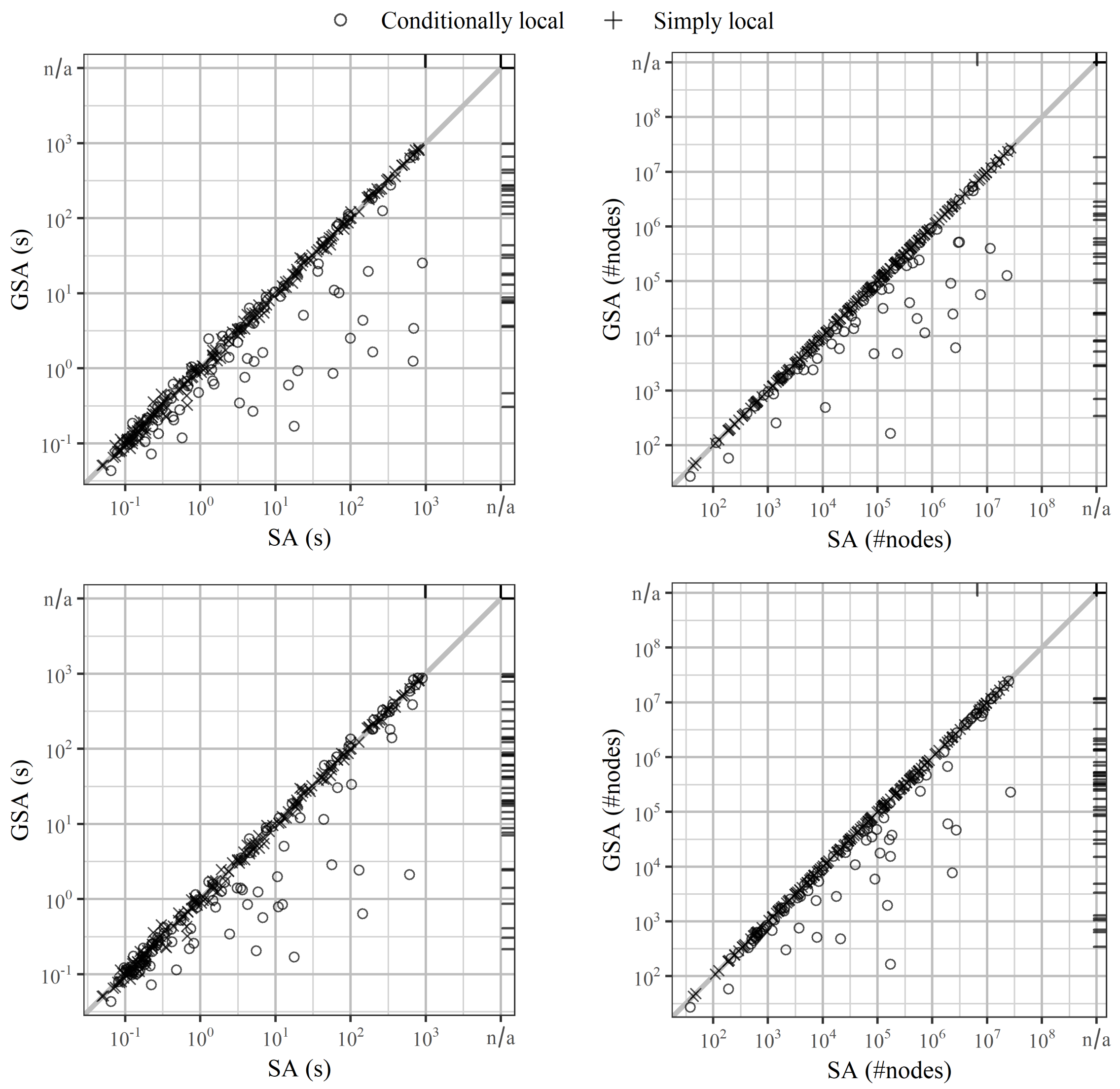


Figure 6.3: Main results of the experiment: running times and total number of created nodes with sloan ordering (top row) and modified sloan ordering (bottom row).

As an auxiliary result and without any illustration, we also report that out of the 117 cases when the sloan ordering and the modified ordering were different and we have data about the final MDD size, the modified sloan ordering produced smaller final MDDs 69 times and larger MDDs 39 times. This motivates further work on variable orderings like in [Amp+18]. We have also compared the SA with sloan ordering and the GSA with the modified sloan ordering to find that the GSA with the modified sloan ordering was better in 78 cases and worse in 16 cases (considering only at least a factor of 2 in both cases).

6.6 Summary and Future Work

In this chapter, I have formally introduced the *generalized saturation algorithm* (GSA), a new saturation algorithm enhanced with the notion of *conditional locality*. I have discussed the effects of using conditional locality and empirically evaluated our approach on Petri nets from the Model Checking Contest to find that the GSA has virtually no overhead compared to the original saturation algorithm, but can outperform it by orders of magnitude when conditional locality can be exploited.

I have made theoretical considerations and prepared the algorithm to be compatible with a wide range of next-state representations as well as the on-the-fly update approach described in [CMS06]. The GSA seems to be superior to the original saturation algorithm on Petri net models, but its efficiency over more general classes of models is yet to be explored.

Thesis 3 I designed an enhancement of the saturation algorithm using the concept of conditional locality. I showed that the classic constrained saturation algorithm, the extension for prioritized Petri nets (Thesis 1), as well as the extension for computing the product state space (Thesis 2) are instances of this generalized saturation algorithm.

3.1 I defined the concept of conditional locality and conditionally saturated nodes, relaxing the notion of locality used in the original saturation algorithm.

3.2 Based on conditional locality, I designed the generalized saturation algorithm (GSA) which enhances the saturation effect and therefore improves the original algorithm. I formally proved the correctness of the new algorithm.

3.3 I showed that the GSA generalizes a family of saturation variants based on constrained saturation. I described the original constrained saturation algorithm along with the extended variants proposed in this work in terms of the GSA.

3.4 I evaluated the GSA on Petri net models of the model checking contest (MCC) and showed that it may outperform the original saturation algorithm by orders of magnitudes in some cases, while it has no considerable overhead in any other case.

The introduction of abstract next-state diagrams and the automatic partitioning of next-state relations in the GSA opens interesting possibilities in potential next-state representations. Behind an ANSD, an *expression diagram* would be an implementation where nodes are quantifier-free first order logic formulas, and node indexing is substitution of a variable. This would be a generalization of implicit relation trees and will be a direct follow-up of this research. With expression diagrams, symbolic transition systems would be directly analyzable with saturation, facilitating the integration of saturation-based model checking and counterexample-guided abstraction refinement [Cla+00], two powerful approaches that could enhance each other even further.

Chapter 7

Summary of the Research Results

7.1 Thesis 1: Analysis of Generalized Stochastic Petri Nets with Symbolic State Space Generation

GSPNs introduce transition priorities into Petri net models. Timed transitions always have less priority than immediate transitions, and the formalism allows the modeler to specify priorities between immediate transitions as well. Therefore, state space exploration should be able to consider the various priority levels of the transitions, leading to the reduced transition locality.

The key idea behind the result of Thesis 1 was that fireability of a transition can be checked based on two separate aspects: *enabledness* as in Petri nets without priorities and the *highest priority* assigned to any (other) enabled transition. For prioritized Petri nets, the highest priority assigned to any enabled transition in any marking can be compiled statically before state space exploration and is independent from the representation of the transitions themselves. To encode this information, I have introduced Edge-Valued Interval Decision Diagrams (EVIDD).

An EVIDD is hybrid between of Edge-valued Decision Diagrams (EDD) [RS10] and Interval Decision Diagrams (IDD) [Tov08]: possible values of a variable are not enumerated but partitioned into intervals (like in IDDs) and each decision amounts to a portion of a value that is computed for the evaluated vector (like in EDDs, as opposed to having only a binary outcome).

It is capable of representing the enabling (and disabling) regions of a transition with its priority as a path in the diagram. The Union-Max operation (implemented by procedure Maximum) is defined recursively on handles and computes an EVIDD representing a disjoint set of regions with maximum priorities as described above. This way, a single EVIDD can encode the highest priority of enabled transitions in any marking.

With an EVIDD describing the highest priority of enabled transitions in every possible marking, fireability can be decided during the computation of the next states in saturation by traversing the EVIDD simultaneously with the decision diagram encoding the state space: if at any point the sum of integer labels exceed the priority of the current transition, it cannot fire and the resulting next state set on that level in the recursion will be empty.

Results. There is not much previous work available on the efficient processing of prioritized Petri nets with saturation. A paper by A. S. Miner et al. [Min01] describes an approach that encodes priorities in the transition relation and uses matrix-diagrams (a special kind of decision diagrams to describe relations) along with a splitting algorithm to repartition the next-state relation such that each partition is as local as possible. My approach has the advantage that the original locality is preserved in the transition relations and the modified saturation algorithm keeps track of priority-related information *for all transitions* in one place, with a compact data structure [c4]. Experimental evaluation showed that this is indeed an advantage and the EVIDD-based solution scales better than that of [Min01].

Thesis 1 I designed an algorithm to help the efficient analysis of Generalized Stochastic Petri Nets (GSPN). It extends the traditional saturation algorithm to perform a more efficient symbolic state space exploration of systems with prioritized transitions.

1.1 I introduced a new type of decision diagram called Edge-Valued Interval Decision Diagram (EVIDD) that can encode enabledness of prioritized transitions in GSPNs.
1.2 I extended the saturation algorithm to handle prioritized transitions efficiently using an EVIDD instead of encoding priorities in the transition relation.
1.3 I evaluated the algorithm and showed that it scales better than previously known approaches.

The importance of these results is that the efficient state space exploration of GSPNs can be a bottleneck of stochastic analysis, so advancements in this field will improve the whole process. Saturation-based state space exploration can handle larger state spaces, and the resulting decision diagram representations can be used to compute efficient decompositions for numerical solvers [c5]. The solvers, in turn, can process these larger state spaces with an acceptable resource budget [c7], leading to scalable stochastic analysis that is necessary to prove extra-functional requirements in safety-critical systems [j1; c6].

7.2 Thesis 2: Saturation-Based Incremental Model Checking for Linear Temporal Logic

Challenge 2 and 3 showed that symbolic LTL model checking has two distinct problems to solve: computing the combined state space of the system and a Büchi automaton describing the negated property, and looking for accepting cycles that represent a counterexample in the system.

Synchronizing with Büchi automata. Computing the combined state space is challenging for saturation because each transition has to be synchronized with one of the transitions of the Büchi automaton such that the transition of the automaton is labeled with an expression that is true in the target state. This implies a read-dependency between variables in the LTL property and all transitions in the system, spoiling locality and reducing the effectiveness of saturation.

The solution for this is based on a similar idea as in Thesis 1. Since the state space and the transition relation of the Büchi automaton are statically available, a data structure encoding the “enabledness” (the satisfaction of expressions on automaton transitions) can be compiled offline. If we assume that atomic propositions in the property are all comparisons between a single state variable and a constant value (and never between two variables), this structure is a finite decision diagram: the upper part consists of levels corresponding to atomic propositions, ordered by the referenced variable, where arcs are labeled by 1 (*true*) and 0 (*false*) (representing the result of evaluating the atomic proposition with a value of the referenced variable), and the lower part is a representation of a subset of the transition relation of the Büchi automaton. Each subset contains the transitions that are labeled according to the evaluation of atomic propositions along the path leading to the lower part.

The saturation algorithm is extended as follows. We assume that the variable encoding the state of the Büchi automaton is on the lowest level. The algorithm will keep evaluating the atomic propositions in the previously described decision diagram until it reaches this bottom level. The transition relation of the model does not specify what happens to the automaton – instead, the reference in the terminal node is used to finish the next-state computation. This way, the information that would spoil the locality of transitions is again handled separately and in one place by the saturation algorithm itself, allowing it to retain the recursive submodel strategy that makes it efficient.

Looking for counterexamples. The problem of finding strongly connected components with saturation has a number of known solutions (e. g. [ZC11]). These assume that the state space is already computed and apply fixed point computations to iteratively remove dead-end states until either no state is left (no SCC) or there are no more dead-ends (the remaining set contains an SCC). Although this would be enough in the LTL model checking setting, but LTL model checkers are traditionally on-the-fly algorithms, i. e. they can stop exploration as soon as a counterexample is found. This is often much faster than processing the whole state space symbolically.

The goal therefore is to apply fixed point computations more often during state space exploration. I proposed an approach that fits the recursive nature of the saturation algorithm: look for SCCs every time a node is saturated, i. e. when a submodel is fully explored, therefore dividing the problem in a similar fashion as saturation does with state space exploration. This approach has a number of advantages. First, it can detect an SCC in an abstract submodel without considering other parts of the system, which will lead to sooner counterexample detection. Secondly, the computation can be incremental in the sense that an SCC must contain at least one firing from a transition that belongs to the current level (otherwise it had been discovered on a lower level), which greatly reduces the search space.

This algorithm still has a considerable overhead and redundancy in the exploration steps. To overcome this, I have proposed two heuristics that can prove the absence of SCCs without an expensive fixed point computation.

Recurring states are states that are reached more than once during exploration. This can be either because there were more than one path leading into the state, or because it is in a cycle which has been closed there. Either way, if there are no recurring states during an exploration, there is no SCC – an observation that can be used as a cheap filter. Recurring states can be efficiently collected in the saturation algorithm and can further restrict the search space of the fixed point algorithm.

A more powerful heuristic is based on *abstractions*. Saturation itself already works on abstract submodels, but those can still be quite large, especially as more and more components are considered. By further abstraction, we can omit all the lower components as well, but this time we will assume that transitions dependent on them are *enabled* (i. e. the omitted places can have any marking – a *may abstraction* – contrary to the higher components not present in the submodel).

I showed that if this abstraction does not have an SCC (which can be efficiently computed by graph algorithms), then there is no SCC in the state space encoded by the node that contains a firing of a transition that belongs to the current level. This is exactly that the fixed point-based algorithm would look for, therefore there is no point in running it. If there is an SCC, fixed point computation can be further limited to transitions whose projections constitute that SCC.

Results. These four components (computation of the combined state space, incremental SCC detection, collection of recurring states and abstractions) constitute an efficient, on-the-fly, incremental symbolic LTL model checking algorithm that outperforms most of the similar tools that were available at the time of its creation [j2; c8]. Extensive measurements on models of the Model Checking Contest showed that the algorithm is often orders of magnitude faster than its competitors, which is a significant step towards scalable LTL model checking for real-life systems.

Thesis 2 I designed an incremental, on-the-fly algorithm for the model checking of properties described by linear temporal logic (LTL), extending the saturation algorithm for state space generation and using it for fixed point computations.

2.1 I extended the saturation algorithm to directly generate the state space of the product system obtained by combining the system-under-analysis and a Büchi automaton describing the LTL property. The advantage of direct generation is that it avoids the explicit computation of the product transition relation.

2.2 I designed an algorithm to incrementally search for strongly connected components (SCC) in the state space during its generation by the saturation algorithm, using the saturation algorithm to compute the necessary fixed points. This approach enables on-the-fly model checking, i. e. the algorithm can terminate as soon as a witness is found.

2.3 I introduced two heuristics that complement the incremental SCC detection algorithm. Using abstraction-based techniques and the concept of recurring states, the heuristics can prove the absence of an SCC and therefore can speed up the search by preventing unnecessary fixed point computations.

2.4 I evaluated the resulting LTL model checking algorithm on models of the Model Checking Contest (MCC), comparing the runtime with three tools that represented the state of the art: the algorithm was found to be often orders of magnitude faster than its competitors.

The importance of these results is that the introduced LTL model checking algorithm often scales better than previously known approaches, while its main ideas are orthogonal to many other improvements that can be found in the literature, promising an even better result when combined. LTL is a popular formalism for specifying temporal properties, especially fairness properties, which cannot be expressed in the other popular formalism, computational-tree logic (CTL). Because of this, advancement in this direction can help in achieving wide-spread use of formal verification in (concurrent) safety-critical systems where fairness is a crucial part or precondition of properties to verify. The algorithm is implemented in the PetriDotNet framework[1] [j1; c6]

7.3 Thesis 3: Enhancement and Generalization of the Saturation Algorithm

Work on the previous theses revealed common problems that appear in many different contexts: *1)* different next-state representations (including but not restricted to different types of decision diagrams) usually come with a specialized variant of the saturation algorithm and are not compatible with each other, as well as *2)* losing locality is generally a concern in every variant, whereas usually there is a workaround that retains some of the locality at least.

According to our experience in implementing saturation-based model checkers, a generic representation of transition relations that works well with saturation would be desirable to separate the algorithm from the representation [c4]. We have introduced Abstract Next-State Diagrams (ANSD) to

[1] http://petridotnet.inf.mit.bme.hu/en/

generalize *decision diagram-like representations* by extracting their common features into an abstract interface.

The core of this thesis is an observation that resulted in the introduction of *conditional locality.* The recursive saturation algorithm exploits the fact that transitions in an abstract submodel belonging to a decision diagram node are independent from the higher variables that are not included in the submodel, therefore their enabledness can be checked and they can be fired arbitrary many times. This is important because computing a local fixed point for the submodel requires the exhaustive firing of all transitions. But can any other transition be fired arbitrary many times?

The answer is yes: for example, transitions with read-only dependencies to higher variables could be fired arbitrary many times, because they will not change the value of the omitted state variables, *given that they are enabled.* In general, any transition that does not change the value of omitted variables can be fired arbitrary many times, even if the value of that variable affects the outcome of the firing.

Conditional locality formulates exactly this property: a transition is conditionally local *with respect to a prefix of a state vector* if firing it from a state with that prefix will result in a state with the same prefix, i. e. those values are not changed [c3]. This means that recursive abstract submodels now contain the places above the current level and also those transitions that have write dependency with the current variable and at most read-only dependency to higher levels.

I have proposed a generalized version of saturation based on conditional locality that computes these submodels dynamically, automatically partitioning the transition relation to process everything on the lowest level possible [c3]. This approach enhances the saturation effect, because a larger portion of the work is performed on small submodels. Even better, including more transitions in the submodels lead to sub-results that are more likely to be final, yielding a faster and more memory-efficient algorithm.

The generalized saturation algorithm (GSA) works with ANSDs to represent any next-state relation. With the automatic partitioning, representations introduced in Theses 1 and 2 (EVIDDs and the decision diagram for the Büchi automaton) as well as previously published variants such as the constraint from constrained saturation can be dynamically intersected with the original transition relation under the ANSD abstraction layer, so the saturation algorithm itself does not have to be adapted. Furthermore, the SCC detection algorithm (along with the heuristics) of Thesis 2 can also benefit from processing transitions on a lower level, as this can speed up the computation and also result in sooner detection of SCCs (on a lower level).

Results. I have compared the GSA with the original saturation algorithm on Petri net models of the MCC to find that it has virtually no overhead when conditional locality is the same as simple locality, whereas it is often orders of magnitude faster and more memory efficient on models with read-only dependencies. The experiments imply that the GSA takes the purely beneficial ideas of saturation one step further without introducing any overhead, while also generalizing the solutions of a family of problems that were hard to solve in the context of the original algorithm. Even though this result covers some of the results in Theses 1 and 2, it is my most recent and most important contribution in this field that would not have been conceived without identifying and generalizing the common ideas in these and other preceding results.

Thesis 3 I designed an enhancement of the saturation algorithm using the concept of conditional locality. I showed that the classic constrained saturation algorithm, the extension for prioritized Petri nets (Thesis 1), as well as the extension for computing the product state space

(Thesis 2) are instances of this generalized saturation algorithm.

3.1 I defined the concept of conditional locality and conditionally saturated nodes, relaxing the notion of locality used in the original saturation algorithm.

3.2 Based on conditional locality, I designed the generalized saturation algorithm (GSA) which enhances the saturation effect and therefore improves the original algorithm. I formally proved the correctness of the new algorithm.

3.3 I showed that the GSA generalizes a family of saturation variants based on constrained saturation. I described the original constrained saturation algorithm along with the extended variants proposed in this work in terms of the GSA.

3.4 I evaluated the GSA on Petri net models of the model checking contest (MCC) and showed that it may outperform the original saturation algorithm by orders of magnitudes in some cases, while it has no considerable overhead in any other case.

The importance of these results is that the saturation algorithm was already one of the most successful symbolic model checking algorithms, which has been improved with the introduction of conditional locality. One of the largest problems in model checking is state space explosion, that can be countered by better scaling and abstractions, which is combined in this result. Furthermore, this result generalizes a family of algorithms, facilitating the correctness, compatibility, a better understanding and maintainable implementation of saturation-based model checking tools by providing a common framework and general correctness proofs for the core algorithm. Finally, the ANSD abstraction layer for the transition relation and the automatic partitioning provided by the GSA enable the direct model checking of more general modeling formalisms (e. g. hierarchical state machines) which in turn facilitates the integration of model checking tools to high-level modeling tools such as the Gamma Statechart Composition Framework (developed by Bence Graics under my supervision) [c12; c13; c14].

7.4 Application of the New Results

7.4.1 Stochastic Analysis of Generalized Stochastic Petri Nets

Saturation for prioritized Petri nets has been used in a tool for the stochastic analysis of GSPNs, where state space exploration is performed by saturation [c4], the state space decomposition is based on decision diagrams [c5] and numerical analysis is performed by a configurable solver framework [c7]. Stochastic analysis is implemented in the PetriDotNet framework[2] [j1; c6] as well, which has been used in the dependability analysis of automotive embedded systems.

7.4.2 LTL Model Checking

The saturation-based on-the-fly, incremental LTL model checking algorithm is also implemented in the PetriDotNet framework that is freely available online. With this, PetriDotNet offers a comprehensive set of analysis algorithms built into a graphical editor, allowing users to model, simulate and analyze Petri nets and stochastic Petri nets using properties expressed either as invariants, CTL or LTL formulas or extra-functional metrics. The tool has been used in a number of independent projects as discussed in [j1]. The LTL model checking algorithm described in the thesis was the first one to successfully verify LTL properties on the so-called PRISE model, which models a safety procedure in the Paks Nuclear Power Plant in Hungary detecting primary-to-secondary leakage accidents [Ném+09].

[2] http://petridotnet.inf.mit.bme.hu/en/

7.4.3 The Generalized Saturation Algorithm

Even though the GSA is fairly new, we are working on exploiting its properties in the Theta Configurable Abstraction Refinement-based Verification Framework [Tót+17] developed by our research group. The framework offers abstraction-based software model checking capabilities with counterexample-guided abstraction refinement (CEGAR), as well as various input languages to facilitate integration with modeling tools. A promising research direction is the combination of saturation and CEGAR. Concurrently, we are working on integrating the framework with the Gamma Statechart Composition Framework [c12; c13; c14], where the verification of component-based systems will benefit from the power of saturation algorithm. The Gamma framework is used in industrial projects for the modeling and analysis of embedded systems as well as for code and test generation.

7.4.4 Use Cases for Model Checking

A part of my research not strictly related to the theses was about the use cases of model checking, which guided my work towards practical applicability. In the CECRIS project[3], I have developed a model checking-based methodology for automated Failure Mode and Effects Analysis (FMEA) of software [j10; c16]. The approach relies on injecting non-deterministically activating faults into the program, then analyzing its behavior with a model checker for different purposes. For FMEA, one is concerned about whether the fault activation will result in a system-level failure, which can be specified separately. The approach can also be used to evaluate fault tolerance mechanisms and error detectors by checking the conformance of the fault-free version and the faulty version coupled with the evaluated mechanism. The results of the project including my work has been published in a book [b17].

In cooperation with my student Levente Bajczi we have also introduced and investigated a novel problem domain involving parallel programs executed on multi-core processors, where the memory controller has a crucial role on the correctness of the system as a whole [c11]. With the widespread use of custom-built processors in embedded systems, faulty memory controllers are less and less rare, party because they tend to be very complex, and also because hardware verification techniques are now able to reveal the problem even after release. Fortunately, these purpose-built chips often run a single program during their lifetime, and the activation of the fault is often linked to specific circumstances. If our program will never encounter those circumstances, then there should be no activation and the problem will remain dormant. Our work showed that today's model checkers are not very well suited for this problem, opening new research questions and challenges with the formal definition of a new use case.

[3]FP7–Marie Curie (IAPP) number 324334

Publications

Publication List

Number of publications:	17
Number of peer-reviewed journal papers (written in English):	4
Number of articles in journals indexed by WoS or Scopus:	4
Number of publications (in English) with at least 50% contribution of the author:	7
Number of peer-reviewed publications:	17
Number of independent citations:	18

Publications Linked to the Theses

	Journal papers	International conference and workshop papers
Thesis 1	[j1]*	[c4]*, [c5], [c6]*, [c7]
Thesis 2	[j1]*, [j2]	[c6]*, [c8]
Thesis 3	–	[c3] [c4]*

* Papers marked by an asterisk belong to more than one theses.
This classification follows the faculty's Ph.D. publication score system.

Journal Papers

[j1] András Vörös, Dániel Darvas, Ákos Hajdu, Attila Klenik, Kristóf Marussy, Vince Molnár, Tamás Bartha, and István Majzik. Industrial applications of the PetriDotNet modelling and analysis tool. *Science of Compututer Programming* 157, 2018, pp. 17–40. DOI: 10.1016/j.scico.2017.09.003. URL: https://doi.org/10.1016/j.scico.2017.09.003.
▷ *The implementation of the PetriDotNet framework is a joint work of the authors, based on the original work of Bertalan Szilvási. The saturation-based LTL model checking algorithm is my own contribution, supervised by A. Vörös and T. Bartha.*

[j2] Vince Molnár, András Vörös, Dániel Darvas, Tamás Bartha, and István Majzik. Component-wise incremental LTL model checking. *Formal Aspects of Computing* 28(3), 2016, pp. 345–379. DOI: 10.1007/s00165-015-0347-x. URL: https://doi.org/10.1007/s00165-015-0347-x.
▷ *Own contribution, joint paper with B.Sc., M.Sc. and Ph.D. supervisors.*

International Conference and Workshop Papers

[c3] Vince Molnár and István Majzik. Saturation enhanced with conditional locality: Application to Petri nets. In: Susanna Donatelli and Stefan Haar (eds.), *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*, Lecture Notes in Computer Science, vol. 11522, pp. 342–361. Springer, 2019. doi: 10.1007/978-3-030-21571-2_19. url: https://doi.org/10.1007/978-3-030-21571-2%5C_19.
▷ *Own contribution, joint paper with Ph.D. supervisor.*

[c4] Kristóf Marussy, Vince Molnár, András Vörös, and István Majzik. Getting the priorities right: Saturation for prioritised Petri nets. In: Wil M. P. van der Aalst and Eike Best (eds.), *Application and Theory of Petri Nets and Concurrency - 38th International Conference, PETRI NETS 2017, Zaragoza, Spain, June 25-30, 2017, Proceedings*, Lecture Notes in Computer Science, vol. 10258, pp. 223–242. Springer, 2017. doi: 10.1007/978-3-319-57861-3_14. url: https://doi.org/10.1007/978-3-319-57861-3%5C_14.
▷ *Own contribution, joint paper with M.Sc. and Ph.D. supervisors and my student K. Marussy.*

[c5] Kristóf Marussy, Attila Klenik, Vince Molnár, András Vörös, István Majzik, and Miklós Telek. Efficient decomposition algorithm for stationary analysis of complex stochastic Petri net models. In: Fabrice Kordon and Daniel Moldt (eds.), *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*, Lecture Notes in Computer Science, vol. 9698, pp. 281–300. Springer, 2016. doi: 10.1007/978-3-319-39086-4_17. url: https://doi.org/10.1007/978-3-319-39086-4%5C_17.
▷ *The stochastic analysis algorithm is the contribution of K. Marussy. The macro-state decomposition algorithm is my own contribution.*

[c6] András Vörös, Dániel Darvas, Vince Molnár, Attila Klenik, Ákos Hajdu, Attila Jámbor, Tamás Bartha, and István Majzik. PetriDotNet 1.5: Extensible Petri net editor and analyser for education and research. In: Fabrice Kordon and Daniel Moldt (eds.), *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*, Lecture Notes in Computer Science, vol. 9698, pp. 123–132. Springer, 2016. doi: 10.1007/978-3-319-39086-4_9. url: https://doi.org/10.1007/978-3-319-39086-4%5C_9.
▷ *The implementation of the PetriDotNet framework is a joint work of the authors, based on the original work of Bertalan Szilvási. The saturation-based LTL model checking algorithm is my own contribution, supervised by A. Vörös and T. Bartha.*

[c7] Kristóf Marussy, Attila Klenik, Vince Molnár, András Vörös, Miklós Telek, and István Majzik. Configurable numerical analysis for stochastic systems. In: *2016 International Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR)*, pp. 1–10. Apr. 2016. doi: 10.1109/SNR.2016.7479383.
▷ *Contribution of my students K. Marussy and A. Klenik, supervised by A. Vörös, M. Telek, I. Majzik and myself.*

[c8] Vince Molnár, Dániel Darvas, András Vörös, and Tamás Bartha. Saturation-based incremental LTL model checking with inductive proofs. In: Christel Baier and Cesare Tinelli (eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, Lecture Notes in Computer Science, vol. 9035, pp. 643–657. Springer, 2015. doi: 10.1007/978-3-662-46681-0_58. url: https://doi.org/10.1007/978-3-662-46681-0%5C_58.

▷ *Own contribution, joint paper with B.Sc., M.Sc. and Ph.D. supervisors.*

Local Events

[l9] Vince Molnár and András Vörös. Synchronous product automaton generation for controller optimization. In: *ASCONIKK 2014: Extended Abstracts. I. Information Technologies for Logistic Systems*, pp. 22–29. Veszprém, Hungary: University of Pannonia, Dec. 2014.
▷ *The tableau automaton-based product computation algorithm is the contribution of A. Vörös and shares ideas with this work.*

Additional Publications (Related to Use Cases and Applications of the Theses)

Journal Papers

[j10] Vince Molnár and István Majzik. Model checking-based software-FMEA: Assessment of fault tolerance and error detection mechanisms. *Periodica Polytechnica Electrical Engineering and Computer Science* 61(2), 2017, pp. 132–150. DOI: https://doi.org/10.3311/PPee.9755. URL: https://pp.bme.hu/eecs/article/view/9755.

International Conference and Workshop Papers

[c11] Levente Bajczi, András Vörös, and Vince Molnár. Will my program break on this faulty processor? - Formal analysis of hardware fault activations in concurrent embedded software. *Transactions on Embedded Computing Systems* 18(5s), 2019, pp. 1–21. DOI: https://doi.org/10.1145/3358238.

[c12] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: Design, verification and code generation for component-based reactive systems. In: Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (eds.), *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 113–116. ACM, 2018. DOI: 10.1145/3183440.3183489. URL: https://doi.org/10.1145/3183440.3183489.

[c13] Bence Graics and Vince Molnár. Mix-and-match composition in the Gamma framework. In: Béla Pataki (ed.), *Proceedings of the 25th PhD Mini-Symposium*, pp. 24–27. Budapest, Hungary, 2018.

[c14] Bence Graics and Vince Molnár. Formal compositional semantics for Yakindu statecharts. In: Béla Pataki (ed.), *Proceedings of the 24th PhD Mini-Symposium*, pp. 22–25. Budapest, Hungary, 2017.

[c15] Vince Molnár and István Majzik. Constraint programming with multi-valued decision diagrams: A saturation approach. In: Béla Pataki (ed.), *Proceedings of the 24th PhD Mini-Symposium*, pp. 54–57. Budapest, Hungary, 2017.

[c16] Vince Molnár and István Majzik. Evaluation of fault tolerance mechanisms with model checking. In: Béla Pataki (ed.), *Proceedings of the 23rd PhD Mini-Symposium*, pp. 30–33. Budapest, Hungary, 2016.

Book Chapters

[b17] Valentina Bonfiglio, Francesco Brancati, Francesco Rossi, Andrea Bondavalli, Leonardo Montecchi, András Pataricza, Imre Kocsis, and Vince Molnár. Composable framework support for software-FMEA through model execution. In: Bondavalli Andrea and Brancati Francesco (eds.), *Certifications of Critical Systems – The CECRIS Experience*, pp. 183–200. Delft, Netherlands: River Publishers, 2017.

Additional Work

[a18] Vince Molnár. Advanced Saturation-based Model Checking. Master's Thesis. Budapest University of Technology and Economics, 2014. URL: https://diplomaterv.vik.bme.hu/en/Theses/Szaturacio-alapu-modellellenorzes.

[a19] Vince Molnár. Szaturáció alapú modellellenőrzés lineáris idejű tulajdonságokhoz [in Hungarian; Saturation-based model checking for linear temporal properties]. Bachelor's Thesis. Budapest University of Technology and Economics, 2013. URL: http://petridotnet.inf.mit.bme.hu/publications/BSThesis2013_Molnar.pdf.

[a20] Vince Molnár and Dániel Segesdi. Múlt és jövő: Új algoritmusok lineáris temporális tulajdonságok szaturáció-alapú modellellenőrzésére [in Hungarian; Future and past: New algorithms for the saturation-based model checking of linear temporal properties]. Scientific Students' Association Report. 1st prize, national 2nd prize (OTDK 2015). 2013. URL: http://petridotnet.inf.mit.bme.hu/publications/TDK2013_MolnarSegesdi.pdf.
▷ *The Past-LTL to Büchi automaton translation and the automaton simplification algorithm is the contribution of D. Segesdi. The automata-theoretic model checking algorithm based on saturation is my own contribution.*

[a21] Vince Molnár. Szaturáció alapú modellellenőrzés lineáris idejű tulajdonságokhoz [in Hungarian; Saturation-based model checking for linear temporal properties]. Scientific Students' Association Report. 2nd prize. 2012. URL: http://petridotnet.inf.mit.bme.hu/publications/TDK2012_Molnar.pdf.

Bibliography

[Ajm88] Marco Ajmone Marsan. Stochastic Petri nets: an elementary introduction. In: Grzegorz Rozenberg (ed.), *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets, held in Venice, Italy in June 1988, selected papers*, Lecture Notes in Computer Science, vol. 424, pp. 1–29. Springer, 1988.

[Amp+18] Elvio Gilberto Amparore, Susanna Donatelli, Marco Beccuti, Giulio Garbi, and Andrew S. Miner. Decision diagrams for Petri nets: A comparison of variable ordering algorithms. *T. Petri Nets and Other Models of Concurrency* 13, 2018, pp. 73–92.

[Ben+14] Ala Eddine Ben Salem, Alexandre Duret-Lutz, Fabrice Kordon, and Yann Thierry-Mieg. Symbolic model checking of stutter invariant properties using generalized testing automata. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 8413, Springer, 2014. DOI: 10.1007/978-3-642-54862-8_38.

[BH14] Dines Bjørner and Klaus Havelund. 40 years of formal methods - some obstacles and some possibilities? In: Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun (eds.), *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, Lecture Notes in Computer Science, vol. 8442, pp. 42–61. Springer, 2014. DOI: 10.1007/978-3-319-06410-9_4.

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In: W. Rance Cleaveland (ed.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, 1999. DOI: 10.1007/3-540-49059-0_14.

[BM19] Shruti Biswal and Andrew S. Miner. Improving saturation efficiency with implicit relations. In: Susanna Donatelli and Stefan Haar (eds.), *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*, Lecture Notes in Computer Science, vol. 11522, pp. 301–320. Springer, 2019. DOI: 10.1007/978-3-030-21571-2_17.

[Bra+11] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In: Per Bjesse and Anna Slobodová (eds.), *Proc. of the International Conference on Formal Methods in Computer-Aided Design*, pp. 144–153. FMCAD Inc, 2011.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* C-35(8), 1986, pp. 677–691. doi: 10.1109/TC.1986.1676819.

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. In: Ernest Nagel, Patrick Suppes, and Alfred Tarski (eds.), *Proc. of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, pp. 1–11. Stanford Univ. Press, 1962.

[Bur+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* 98(2), 1992, pp. 142–170. doi: 10.1016/0890-5401(92)90017-A. (Visited on 09/08/2014).

[BZC99] Armin Biere, Yunshan Zhu, and Edmund M. Clarke. Multiple state and single state tableaux for combining local and global model checking. In: Ernst-Rüdiger Olderog and Bernhard Steffen (eds.), *Correct System Design*, Lecture Notes in Computer Science, vol. 1710, pp. 163–179. Springer, 1999.

[Cav+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Alberto Griggio, Marco Roveri, and Stefano Tonetta. *The nuXmv Symbolic Model Checker.* Tech. rep. Fondazione Bruno Kessler, 2014.

[CCY06] Ming-Ying Chung, Gianfranco Ciardo, and Andy Jinqing Yu. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In: Susanne Graf and Wenhui Zhang (eds.), *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006.* Lecture Notes in Computer Science, vol. 4218, pp. 51–66. Springer, 2006. doi: 10.1007/11901914_7.

[CGH97] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design* 10(1), 1997, pp. 47–71. doi: 10.1023/A:1008615614281.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[Chi+93] Giovanni Chiola, Marsan Marco Ajmone, Gianfranco Balbo, and Gianni Conte. Generalized stochastic Petri nets: A definition at the net level and its implications. *IEEE Trans. Software Eng.* 19(2), 1993, pp. 89–107.

[Cia+06] G. Ciardo, R. L. Jones III, A. S. Miner, and R. I. Siminiceanu. Logic and stochastic modeling with SMART. *Perform. Eval.* 63(6), 2006, pp. 578–608. doi: 10.1016/j.peva.2005.06.001.

[Cim+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: an open-source tool for symbolic model checking. In: Ed Brinksma and Kim G. Larsen (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2404, pp. 359–364. Springer, 2002. url: http://dx.doi.org/10.1007/3-540-45657-0_29.

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In: E. Allen Emerson and A. Prasad Sistla (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer, 2000. doi: 10.1007/10722167_15.

[Cla+96] Edmund M. Clarke, Kenneth L. McMillan, Sergio V. Campos, and Vasiliki Hartonas-Garmhausen. Symbolic model checking. In: Rajeev Alur and Thomas A. Henzinger (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1102, pp. 419–422. Springer, 1996.

[CLS01] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: an efficient iteration strategy for symbolic state-space generation. In: *Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 328–342. 2001.

[CMS03] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In: Hubert Garavel Hubert.Garavel and John Hatcliff (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, vol. 2619, pp. 379–393. Springer, 2003.

[CMS06] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer* 8(1), 2006, pp. 4–25. DOI: 10.1007/s10009-005-0188-7. (Visited on 08/24/2014).

[Coh91] Joëlle Cohen-Chesnot. On the expressive power of temporal logic for infinite words. *Theor. Comput. Sci.* 83(2), 1991, pp. 301–312. DOI: 10.1016/0304-3975(91)90281-6.

[Cou+91] Costas A. Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory efficient algorithms for the verification of temporal properties. In: Edmund M. Clarke and Robert P. Kurshan (eds.), *Computer-Aided Verification*, Lecture Notes in Computer Science, vol. 531, pp. 233–242. Springer, 1991. URL: http://dx.doi.org/10.1007/BFb0023737.

[CT93] Gianfranco Ciardo and Kishor S. Trivedi. A decomposition approach for stochastic reward net models. *Perform. Eval.* 18(1), 1993, pp. 37–59. DOI: 10.1016/0166-5316(93)90026-Q.

[CZJ09] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? In: Lubos Brim and Jaco van de Pol (eds.), *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009.* EPTCS, vol. 14, pp. 1–17. 2009. DOI: 10.4204/EPTCS.14.1.

[DH19] Susanna Donatelli and Stefan Haar, eds. *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings.* Vol. 11522. Lecture Notes in Computer Science. Springer, 2019. DOI: 10.1007/978-3-030-21571-2.

[DMP19] Tom van Dijk, Jeroen Meijer, and Jaco van de Pol. Multi-core on-the-fly saturation. In: Tomás Vojnar and Lijun Zhang (eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, Lecture Notes in Computer Science, vol. 11428, pp. 58–75. Springer, 2019. DOI: 10.1007/978-3-030-17465-1_4.

[DP04] Alexandre Duret-Lutz and Denis Poitrenaud. SPOT: an extensible model checking library using transition-based generalized Büchi automata. In: *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pp. 76–83. 2004. DOI: 10.1109/MASCOT.2004.1348184.

[Dur+11a] Alexandre Duret-Lutz, Kaïs Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Combining explicit and symbolic approaches for better on-the-fly LTL model checking. *CoRR* abs/1106.5700, 2011. http://arxiv.org/abs/1106.5700. URL: http://arxiv.org/abs/1106.5700 (visited on 08/24/2014).

[Dur+11b] Alexandre Duret-Lutz, Kaïs Klai, Denis Poitrenaud, and Yann Thierry-Mieg. Self-loop aggregation product – A new hybrid approach to on-the-fly LTL model checking. In: Tevfik Bultan and Pao-Ann Hsiung (eds.), *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, vol. 6996, pp. 336–350. Springer, 2011. doi: 10.1007/978-3-642-24372-1_24.

[EC80] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In: J. W. de Bakker and Jan van Leeuwen (eds.), *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, Lecture Notes in Computer Science, vol. 85, pp. 169–181. Springer, 1980. doi: 10.1007/3-540-10003-2_69.

[Ger+95] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In: Piotr Dembinski and Marek Sredniawa (eds.), *Proc. of the International Symposium on Protocol Specification, Testing and Verification*, pp. 3–18. Chapman & Hall, Ltd., 1995.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In: Gérard Berry, Hubert Comon, and Alain Finkel (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2102, pp. 53–65. Springer, 2001. doi: 10.1007/3-540-44585-4_6.

[God14] Patrice Godefroid. May/must abstraction-based software model checking for sound verification and falsification. In: Orna Grumberg, Helmut Seidl, and Maximilian Irlbeck (eds.), *Software Systems Safety*, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 36, pp. 1–16. IOS Press, 2014. doi: 10.3233/978-1-61499-385-8-1.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer, 1996.

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In: *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 58–70. ACM, 2002. doi: 10.1145/503272.503279.

[HIK04] Serge Haddad, Jean-Michel Ilié, and Kaïs Klai. Design and evaluation of a symbolic and abstraction-based model checker. In: Farn Wang (ed.), *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, vol. 3299, pp. 196–210. Springer, 2004. doi: 10.1007/978-3-540-30476-0_19. (Visited on 10/03/2014).

[Hil+09] Lom M. Hillah, Ekkart Kindler, Fabrice Kordon, Laure Petrucci, and Nicolas Treves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Petri Net Newsletter* 76, 2009, pp. 9–28. url: http://www.pnml.org/papers/pnnl76.pdf.

[HPY97] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search. In: Gerard J. Holzmann Jean-Charles Grégoire and Doron A. Peled (eds.), *The Spin Verification System*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 81–89. AMS, 1997.

[HTK08] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In: Kees M. van Hee and Rüdiger Valk (eds.), *Applications and Theory of Petri Nets*, Lecture Notes in Computer Science, pp. 211–230. Springer, 2008. url: http://link.springer.com/chapter/10.1007/978-3-540-68746-7_16 (visited on 08/24/2014).

[Kor+18] F. Kordon et al. Complete Results for the 2018 Edition of the Model Checking Contest. http://mcc.lip6.fr/2018/results.php. 2018. (Visited on 2018).

[KP08] Kaïs Klai and Denis Poitrenaud. MC-SOG: an LTL model checker based on symbolic observation graphs. In: Kees M. van Hee and Rüdiger Valk (eds.), *Applications and Theory of Petri Nets*, Lecture Notes in Computer Science, vol. 5062, pp. 288–306. Springer, 2008. DOI: 10.1007/978-3-540-68746-7_20.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica* 16(1963), 1963, pp. 83–94.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer* 26(7), 1993, pp. 18–41.

[Mar+16b] Kristóf Marussy, Attila Klenik, Vince Molnár, András Vörös, Miklós Telek, and István Majzik. Configurable numerical analysis for stochastic systems. In: *2016 International Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR)*, pp. 1–10. 2016. DOI: 10.1109/SNR.2016.7479383.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In: Warren A. Hunt, Jr. and Fabio Somenzi (eds.), *Lecture Notes in Computer Science*, vol. 2725, pp. 1–13. 2003. DOI: 10.1007/978-3-540-45069-6_1.

[McM92] Kenneth L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. UMI Order No. GAX92-24209. PhD thesis. Carnegie Mellon University, 1992.

[MD98] D. Michael Miller and Rolf Drechsler. Implementing a multiple-valued decision diagram package. In: *Proc. of the 28th IEEE International Symposium on Multiple-Valued Logic*, pp. 52–57. 1998. DOI: 10.1109/ISMVL.1998.679287.

[Mei+14] Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. Read, write and copy dependencies for symbolic model checking. In: *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, pp. 204–219. 2014.

[Min01] Andrew S Miner. Efficient solution of GSPNs using canonical matrix diagrams. In: *Petri Nets and Performance Models, 2001. Proceedings. 9th International Workshop on*, pp. 101–110. 2001.

[Min04] Andrew S. Miner. Implicit GSPN reachability set generation using decision diagrams. *Perform. Eval.* 56(1-4), 2004, pp. 145–165. DOI: 10.1016/j.peva.2003.07.005.

[Min06] Andrew S. Miner. Saturation for a general class of models. *IEEE Trans. Software Eng.* 32(8), 2006, pp. 559–570. DOI: 10.1109/TSE.2006.81.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer, 1992.

[Mur89] Tadao Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE* 77(4), 1989, pp. 541–580. DOI: 10.1109/5.24143.

[Ném+09] Erzsébet Németh, Tamás Bartha, Csaba Fazekas, and Katalin M. Hangos. Verification of a primary-to-secondary leaking safety procedure in a nuclear power plant using coloured Petri nets. *Reliability Engineering & System Safety* 94(5), 2009, pp. 942–953. DOI: 10.1016/j.ress.2008.10.012.

[Pel98] Doron Peled. Ten years of partial order reduction. In: Alan J. Hu and Moshe Y. Vardi (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 1427, pp. 17–28. Springer, 1998. URL: http://dx.doi.org/10.1007/BFb0028727.

[Pnu77] Amir Pnueli. The temporal logic of programs. In: *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46–57. IEEE Computer Society, 1977. DOI: 10.1109/SFCS.1977.32.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In: Mariangiola Dezani-Ciancaglini and Ugo Montanari (eds.), *International Symposium on Programming*, pp. 337–351. Springer Berlin Heidelberg, 1982.

[RS10] Pierre Roux and Radu Siminiceanu. Model checking with edge-valued decision diagrams. In: *Proc. of the 2nd NASA Formal Methods Symposium*, pp. 222–226. 2010.

[SB14] Marcin Szpyrka and Agnieszka and Biernacki Jerzy Biernacka. Methods of translation of Petri nets to NuSMV language. In: Louchka Popova-Zeugmann (ed.), *Concurrency, Specification and Programming*, CEUR Workshop Proceedings, vol. 1269, pp. 245–256. 2014.

[Sch03] Philippe Schnoebelen. The complexity of temporal logic model checking. In: *Advances in Modal Logic*, vol. 4, pp. 1–44. King's College Publications, 2003.

[SRB02] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic SCC hull algorithms. In: Mark D. Aagaard and John W. O'Leary (eds.), *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, vol. 2517, pp. 88–105. Springer, 2002. URL: http://link.springer.com/chapter/10.1007/3-540-36126-X_6 (visited on 10/05/2014).

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In: Warren A. Hunt, Jr. and Steven D. Johnson (eds.), *Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer, 2000. DOI: 10.1007/3-540-40922-X_8.

[STV05] Roberto Sebastiani, Stefano Tonetta, and Moshe Y. Vardi. Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking. In: Kousha Etessami and Sriram K. Rajamani (eds.), *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 3576, pp. 350–363. Springer, 2005.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2), 1972, pp. 146–160. DOI: 10.1137/0201010.

[TFP03] Enrique Teruel, Giuliana Franceschinis, and Massimiliano De Pierro. Well-defined generalized stochastic Petri nets: A net-level method to specify priorities. *IEEE Trans. Software Eng.* 29(11), 2003, pp. 962–973.

[Tót+17] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: A framework for abstraction refinement-based model checking. In: Daryl Stewart and Georg Weissenbacher (eds.), *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pp. 176–179. IEEE, 2017. DOI: 10.23919/FMCAD.2017.8102257.

[Tov08] Alexey A. Tovchigrechko. Efficient symbolic analysis of bounded Petri nets using interval decision diagrams. PhD thesis. Brandenburg University of Technology, Cottbus-Senftenberg, Germany, 2008.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In: Faron Moller and Graham Birtwistle (eds.), *Logics for Concurrency*, Lecture Notes in Computer Science, vol. 1043, pp. 238–266. Springer, 1996. DOI: 10.1007/3-540-60915-6_6.

[Vör+11] András Vörös, Tamás Szabó, Attila Jámbor, Dániel Darvas, Ákos Horváth, and Tamás Bartha. Parallel saturation based model checking. In: *10th International Symposium on Parallel and Distributed Computing, ISPDC 2011, Cluj-Napoca, Romania, July 6-8, 2011*, pp. 94–101. IEEE Computer Society, 2011. DOI: 10.1109/ISPDC.2011.23.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In: *Proc. of the Symposium on Logic in Computer Science*, pp. 332–344. IEEE Computer Society, 1986.

[Wan+06] Chao Wang, Roderick Bloem, Gary D. Hachtel, Kavita Ravi, and Fabio Somenzi. Compositional SCC analysis for language emptiness. *Formal Methods in System Design* 28(1), 2006, pp. 5–36. DOI: 10.1007/s10703-006-4617-3.

[WL91] C. Murray Woodside and Yao Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In: *Proc. of 4th Int. Workshop on Petri Nets and Performance Models*, pp. 64–73. 1991. DOI: 10.1109/PNPM.1991.238781.

[ZC09] Yang Zhao and Gianfranco Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In: *Proc. of the 7th Int. Conf. Automated Technology for Verification and Analysis*, pp. 368–381. 2009.

[ZC11] Yang Zhao and Gianfranco Ciardo. Symbolic computation of strongly connected components and fair cycles using saturation. en. *Innovations in Systems and Software Engineering* 7(2), 2011, pp. 141–150. DOI: 10.1007/s11334-011-0146-3. (Visited on 08/24/2014).