



Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Department of Measurement and Information Systems

Incremental extension of the saturation algorithm-based bounded model checking of Petri nets

MASTER'S THESIS

Author

Dániel Darvas

Supervisors

dr. Tamás Bartha,
András Vörös

May 15, 2014

(last fix: August 7, 2014)

Contents

Kivonat	6
Abstract	7
Résumé	8
Introduction	9
1 Background	13
1.1 Model checking	13
1.1.1 Bounded model checking	15
1.1.2 Symbolic model checking	15
1.1.3 Computation Tree Logic	16
1.2 Petri nets	18
1.3 Decision diagrams	20
1.3.1 Binary Decision Diagrams	20
1.3.2 Multivalued Decision Diagrams	21
1.3.3 Edge-valued Decision Diagrams	24
2 Saturation	26
2.1 Overview of the saturation-based algorithms	26
2.1.1 Definitions of model and states	27
2.1.2 Data structures	27
2.1.3 Advantages of saturation	31
2.2 The background of saturation	31

2.3	Challenges	33
2.4	Unbounded state space generation	35
2.4.1	Classic method – the base algorithm	35
2.4.2	Constrained method	36
2.5	Unbounded CTL model checking	36
2.6	Bounded state space generation	38
2.6.1	Truncating	39
2.6.2	Iterative bounded state space exploration	40
2.7	Bounded CTL model checking	41
3	Incremental bounded state space exploration algorithms	42
3.1	What data can be kept?	43
3.2	Restarting algorithm	43
3.3	Continuing algorithm	45
3.3.1	Handling set of initial states	46
3.4	Compacting saturation	47
3.4.1	Main idea of compacting saturation	48
3.4.2	Overview of the challenges	48
3.4.3	Computing the border of the state space	49
3.4.4	Avoiding the redundant computations	50
3.4.5	How does the compacting saturation work?	54
3.4.6	Possible improvement of the compacting saturation	54
3.5	General problems of saturation-based iterative model checking	56
4	Realization of the compacting saturation	58
4.1	PetriDotNet framework	58
4.2	Details of the implementation	59
4.2.1	Implementation of the decision diagrams	60
4.2.2	Creating and destroying node objects	60
4.2.3	Effects of micro-optimization	62
4.2.4	Usage of delegates	68

5	Evaluation	70
5.1	Run time measurements	72
5.1.1	Full state space exploration	73
5.1.2	Evaluation of CTL expressions	74
5.1.3	Scalability	77
5.2	Memory consumption measurements	79
5.2.1	Peak memory consumption	79
5.2.2	Memory allocation measurements	80
5.3	Comparison of compacting and noncompacting methods	81
5.4	Industrial case studies	82
5.4.1	Verification of the PRISE logic in a nuclear power plant	82
5.4.2	Verification of PLC programs in CERN	89
5.5	Conclusions of measurements	91
6	Summary	93
6.1	Future work	94
	Acknowledgement	95
	List of figures	97
	List of algorithms	98
	Bibliography	99
A	Abbreviations and used symbols	105
A.1	Symbols	105
A.2	Abbreviations	106

B	Models	108
B.1	Dining philosophers	108
B.1.1	Phil-N model	108
B.1.2	DPhil-N model	109
B.2	Slotted ring protocol	109
B.3	Manufacturing systems	110
B.4	Tower of Hanoi	112
B.5	Queen	112
B.6	Round robin	113
B.7	Counter	113
C	Pseudocodes	115
C.1	General unbounded saturation	115
C.2	Unified unbounded saturation	117
C.3	Unified bounded saturation	119
C.3.1	Bounded saturation as a module	119
C.4	Restarting bounded saturation	123
C.5	Continuing bounded saturation	123
C.6	Compacting bounded saturation	124
D	Details of measurements	125
D.1	Measurements of implementation details	125
D.2	Measurement tools	126
D.3	Computer used for the measurements	127

HALLGATÓI NYILATKOZAT

Alulírott *Darvas Dániel*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózataán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2014. május 15.

Darvas Dániel
hallgató

Kivonat

Napjainkban a biztonságkritikus alkalmazásoknál és azokon kívül is egyre elterjedtebb a formális módszerek, mint például a modellellenőrzés használata a minőség és megbízhatóság növelése érdekében. Azonban a komplex rendszerek vizsgálata az erőforrások korlátoossága miatt nehéz feladatnak bizonyul. Emiatt számos új modellellenőrző algoritmus látott napvilágot. Közülük egy a szaturációs algoritmus, amely kompakt adatrepresentációjának és speciális iterációs stratégiájának köszönhetően igen hatékornynak bizonyult, különösen aszinkron rendszerek vizsgálata esetén.

A szaturációs algoritmusnak létezik korlátos állapottér-felderítést megvalósító variánsa, amely az ún. sekély problémák vizsgálata esetén ideális. Azonban az eddigi, ezen alapuló korlátos modellellenőrző algoritmus csak neminkrementális módon (minden iterációban a folyamatot teljesen újakezdve) képes működni.

Diplomatervemben megvizsgálom a Petri-hálókon alapuló modellellenőrzés valamint a szaturációs algoritmusok elméleti alapjait. Áttekintem az ide vonatkozó szakirodalmat és ez alapján a diplomatervemhez szükséges alapismeretek kivonatát, amely bemutatja a modellellenőrzés, a Petri-hálók, a döntési diagramok és a szaturációalapú algoritmusok alapjait.

Megvizsgálom továbbá a korábban publikált főbb szaturációalapú korlátos és nemkorlátos állapottér-felderítő algoritmusokat, illetve modellellenőrzési megoldásokat. Áttekintem a szaturációalapú algoritmusok közös vonásait, közös alapelveit, illetve a megvalósítás kihívásait. Megvizsgálom ezeken kívül a különböző szaturációalapú megoldások újdonságait, illetve olyan vonásait, amelyek segítségével a korlátos inkrementális modellellenőrző algoritmusok megtervezhetők.

Munkám során két új, inkrementális modellellenőrző algoritmust terveztem meg (a „folytató” és „kompaktáló” algoritmusokat), amelyek különböző módon képesek inkrementális modellellenőrzést végezni. A megtervezett algoritmusokat implementáltam, majd mérésekkel hasonlítottam össze teljesítményüket különböző esetekre.

Abstract

Nowadays, in safety critical and other applications the usage of formal methods, like model checking is more and more common in order to increase the quality. However, the analysis of complex systems is a difficult task due to the existing resource limitations. This problem inspired many new model checking algorithms. One of them is the saturation-based algorithm which used to be efficient, especially for asynchronous systems, thanks to its compact data representation and special iteration strategy.

There exists a variant of the saturation-based algorithm that performs bounded state space exploration, which can be efficiently used to detect shallow problems. Nevertheless, the existing bounded model checker algorithm cannot run incrementally (therefore every iteration will be restarted completely).

In my thesis, I examine the theoretical background of Petri-net-based model checking and saturation-based algorithms: like model checking, Petri nets, decision diagrams and saturation-based algorithms.

I examine the previously published saturation-based bounded and unbounded state space exploration algorithms and model checking methods. I analyse the similarities and the challenges of the saturation-based algorithms. Moreover, I examine the new ideas of the algorithms and the properties that can be reused to be able to develop bounded incremental model checking algorithms.

In my work I designed two incremental model checking algorithms (the “continuing” and the “compacting” algorithms) which are two different ways to perform incremental model checking. I have also implemented these algorithms and compared their performance by measurements.

Résumé

De nos jours, l'application des méthodes formelles, tel que le model checking, est de plus en plus de courant dans le domaine de systèmes critiques, mais les ressources de calcul limitées font l'analyse des systèmes complexes une tâche difficile. Ce problème a inspiré de nombreux nouveaux algorithmes de model checking. L'un d'entre eux est l'algorithme dit « saturation » qui est efficace, notamment pour les systèmes asynchrones grâce à sa représentation compacte des données et sa stratégie d'itération particulière.

Il existe une variante de l'algorithme saturation qui effectue l'exploration de l'espace d'état bornée qui peut être utilisée efficacement afin de détecter des problèmes à faible profondeur. Néanmoins, cet algorithme ne peut pas fonctionner de façon incrémentale, conduisant toutes les itérations à être redémarrées complètement. Dans ce mémoire, j'étudie la théorie du model checking des réseaux de Petri et des algorithmes saturation, comme le model checking, les réseaux de Petri, les diagrammes de décision et les algorithmes basés sur la saturation.

J'étudie les algorithmes bornés et non bornés déjà publiés pour explorer et vérifier les espaces d'état basée sur la saturation. J'analyse les similitudes et les problèmes des algorithmes basés sur la saturation. Puis, j'examine les idées nouvelles et les particularités de ces algorithmes qui peuvent être réutilisées dans le développement des algorithmes du model checking bornées incrémentielles.

Dans mon travail, j'ai conçu deux algorithmes de model checking (les algorithmes « continu/continuing » et « compactage/compacting ») qui sont deux manières différentes pour effectuer du model checking incrémental. Également j'ai mis en œuvre ces algorithmes et j'ai comparé leurs performances en terme de temps et de mémoire vive utilisée.

Introduction

“Clearly, the need for reliable hardware and software systems is critical. As the involvement of such systems in our lives increases, so too does the burden for ensuring their correctness. Unfortunately, it is no longer feasible to shut down a malfunctioning system in order to restore safety. We are very much dependent on such systems for continuous operation; in fact, in some cases, devices are less safe when they are shut down.” [20]

First cars had completely mechanic steering systems. Then power steering was introduced to help the driver steer the vehicle. The new power steering systems are smarter and smarter and nowadays computer-controlled electrical power steering systems are assembled to the high quality cars. Also, there are working concept cars for the “steer-by-wire” solutions, where the mechanic connection is eliminated between the steering wheel and the wheels. And recently, Google (among others) introduced self-driving cars running automatically in the traffic, without any human interaction.

Road transportation is not the only area where this trend can be observed. The focus from human controlling and mechanic connections moves to computer controlling and electric connections. Therefore the computers provide more and more critical functions and for that reason, the need for high quality, error free systems is bigger than ever.

The quality of the hardware and software systems can be risen using various solutions. One of the most spread of these solutions is the usage of *formal methods*, for example the application of model checking formal verification method. (As the success of the formal methods is the main motivation of my thesis, a longer introduction can be read below.)

My thesis focuses on a family of model checking algorithms, on the so-called *saturation-based algorithms*. The saturation-based algorithms are model checking algorithms providing formal methods to verify the correctness of systems. These algorithms have already proved their efficiency but that does not mean that they cannot be improved. The goal of my thesis is to improve the bounded variants of saturation-based algorithms by enabling them to work incrementally.

Formal methods, formal verification – introduction and motivation

Formal methods are mathematical-based methods that can help to improve the quality of the result of development.

A possible way to define formal methods is the following (quoted from the Encyclopedia of Software Engineering [38]):

Definition 1 (Formal methods). Formal methods used in developing computer systems are mathematically based techniques for describing system properties. [...] A method is *formal* if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and, more relevant, specification, implementation, and correctness. ■

We can view formal methods as a branch of mathematics that may have real world applications, but also we can think about it as a software or hardware engineering method which aims to create better systems [36].

Formal methods are mainly used in the development of safety-critical systems (e.g., in medical systems, transportation, nuclear power plants, etc.). In the case of systems which can cause injury or death, the proof of correctness is extremely important. However, formal methods are useful in other cases too. According to A. Hall [33], “Formal methods should be used wherever the cost of failure is high.” This includes safety-critical systems, embedded systems (where the cost of software update is high), and systems used in large quantity.

While Hall’s mentioned idea was written in 1990, we still cannot see in 2013 that formal methods are used in every case when “the cost of failure is high”. A recent survey of J. Woodcock et al. [52] shows an increasing usage of formal methods, but it is still not a common part of the software or hardware development processes.

The field of formal methods is a controversial topic. There are plenty of common myths about it and many people still do not believe in its usefulness [36]. While it is true that formal methods do not solve all problems related to software and hardware engineering and they have serious limitations, formal methods have proved their usability in many cases.

Hall has written in [33] about the myth that formal methods are not usable in real, large scale informatics projects. As the myth still stands in 2013, I would like to mention some real world projects that used formal methods successfully.

Formal methods were used in the following projects:

- *Météor metro line* Météor (line 14) is a driverless automatic subway line (in Paris, France) opened in 1998. The safety critical parts (the automatic pilot and signalling

subsystem, called PA-SIG) were developed using B Method. No bugs were found after the proof of the models during the integration tests, on-site tests or in operation – even if no unit tests were performed. [4, 1] The B Method was used in other projects related to the Paris subway as well. [37]

- *ERTMS* The safety logic (Logica Di Sicurezza, LDS) of the worldwide-used Level 2 European Railways Train Management System (ERTMS) was verified and validated using formal methods. This is a highly safety-critical system as it controls the train spacing in the ERTMS, therefore it is responsible for keeping a secure distance between the trains. They used various verification techniques: bounded model checking, temporal induction, CEGAR, and software model checkers [17]. Similar problem has been verified by model checking in [39].
- *Remote Agent* The Remote Agent spacecraft control system was used by NASA on the Deep Space 1 mission (launched in 1998, ended in 2001). This controller was able to control the spacecraft without human supervision. A part of the controller was modelled and verified by model checking.

Soon after the launching, a deadlock occurred in a non-verified component. Later, it turned out that with the same model checking techniques they used for other parts, this error could have been found before launching. [34]

- *Intel processors* Intel started to use formal verification in 1995. They started with proving selected areas of CPUs that are new and providing complex functionality. Since then, parts of the microcode are also verified. According to the reports of lead CPU teams, “high quality bugs” can be found using formal methods [31]. And the evolution of formal verification of CPUs is still going on: during the development of the CoreTM i7 processors, the engineers of Intel used formal verification instead of testing to check the execution engine of the CPU. This method proved that even complex hardware systems can be handled by formal verification. Some bugs were missed due to the incorrect formal specification or too late verification, but this is a small amount compared to the former results [35].
- *Nuclear power plants* Formal methods were also involved in the development of nuclear power plants. For instance, Rolls-Royce and Associates have applied formal methods as early as 1988. After the formal verification, there was a testing phase, but the majority of the new bugs were mistakes in the testing code. [8]

As it can be seen, there are multiple fields where formal methods can be applied successfully. The reader can find other examples in [8, 52].

One of the most spread formal verification method is the *model checking*. Model checking enables us to determine whether given formalized requirements are fulfilled by a given formalized model or not.

While in theory, model checking is a handful tool for system verification, in practice, it faces a big problem: the scalability. Even the verification of relatively small systems

can require enormous amount of time and memory, and the computation needs can grow exponentially. This problem induces the development of newer and newer model checking methods and this is the motivation of this thesis too: to improve the existing model checking algorithms, to enlarge the set of analysable problems.

Structure of the thesis

This thesis is structured as follows. Chapter 1 overviews the necessary background, like model checking, the used modelling languages and data structures. After, the saturation-based techniques and former saturation algorithms are presented in Chapter 2. The main contribution of my thesis starts from Chapter 3 where the new, incremental algorithms are discussed. Chapter 4 is dedicated to the realization and implementation of the algorithms. Finally, Chapter 5 shows measurements and evaluates the presented algorithms. Chapter 6 summarizes my work.

In the Appendix, the reader finds a table of symbols and abbreviations, the definition of the models used for benchmarking, and the pseudocodes of the mentioned algorithms.

Chapter 1

Background

This chapter provides an introduction on the background of my thesis. As the goal of this work is to improve the former saturation-based model checking methods, first I introduce and overview the model checking as a technique (Section 1.1). Then, Section 1.2 describes the Petri nets, the modelling method I used to formalize the input of the model checking. Section 1.3 introduces several types of decision diagrams that are used as main data structures and that have key roles in my algorithms.

This chapter contains translations from my earlier works written in Hungarian [21, 24, 25].

1.1 Model checking

Model checking is a formal verification technique for verifying finite state systems. This method was first described by Edmund Clarke and Allen Emerson in [19], also in parallel by Joseph Sifakis [46].

They combined the state space exploration and the temporal logic in order to be able to verify concurrent programs against the given specification. In this method, a requirement is checked on a given model, and the result is whether the model fulfils the requirement or not.

Formally, the definition of model checking problem is as follows (from [18]):

Definition 2 (Model checking problem). Let M be a Kripke structure (i. e., state-transition graph). Let f be a formula of temporal logic (i. e., the specification or property to be checked). Find all states s of M such that $M, s \models f$. ■

So basically, the goal is to find the $\{s \in \mathcal{S} \mid M, s \models f\}$ set (where \mathcal{S} is the set of possible states in M , and the meaning of $M, s \models f$ is “the behaviour of the model M starting from state s (as an initial state) satisfies the property f ”).

Usually, the model checking problem is more specific, as the goal is to decide whether f is true for the initial state s_0 of the model M or not [20]. Therefore the question is: $s_0 \stackrel{?}{\in} \{s \in \mathcal{S} | M, s \models f\}$ ¹.

Formalisms. There are multiple formalism to express the model and the requirement. In this thesis, I will use Petri nets (see Section 1.2) to express the models and CTL (Computation Tree Logic, see Section 1.1.3) to express the requirements, as my goal is to improve the existing Petri net-based CTL model checking methods.

Advantages and disadvantages. Model checking has some advantages and disadvantages. The main advantages are the following [20]:

- This is an automatic method, the user does not need to have deep domain knowledge.
- If a requirement fails, it is often possible to find a counterexample which is a huge help to identify and correct the problems.

However, there are some challenges in model checking, of which the most important is the so-called *state space explosion*. This occurs when the analysed model is complex and its state space contains enormous number of states. Edmund Clarke, one of the founders of model checking has written in [18]:

“State explosion is a major problem. This is absolutely true. The number of global system states of a concurrent system with many processes or complicated data structures can be enormous. All Model Checkers suffer from this problem. In fact, the state explosion problem has been the driving force behind much of the research in Model Checking and the development of new Model Checkers.”

This high memory and computational need led to more and more sophisticated model checking algorithms. There are multiple possible directions to create algorithms that can handle more complex verification problems. Two of them will be introduced in the following: symbolic model checking and bounded model checking. Basically, symbolic model checking tries to lower the memory usage (and also the computational need) by using compact data structures, while bounded model checking tries to reduce the size of the explored part of the state space. These approaches can be combined: in my thesis, my goal is to develop better bounded symbolic model checking algorithms.

¹If multiple initial states exist, the question is more precisely the following: $s_0 \cap \{s \in \mathcal{S} | M, s \models f\} \stackrel{?}{\neq} \emptyset$.

1.1.1 Bounded model checking

A traditional approach for model checking consists of two phases: a state space exploration phase and a temporal logic expression checking phase. In many cases, the exploration of the full state space is not needed or not possible. Consider a reachability problem, in other words, check the truth of the following: at least one state is reachable from the initial state where p is true (in CTL: $\text{EF } p$). If p holds for a state which is reachable within few intermediate states, it is unnecessary to explore the whole state space, as only a small part of the state space (near to the initial state) is enough to decide the truth of the formula.

The *bounded model checking* method allows us to check properties on a k -bounded part of the state space. The k -bounded part of a state space is $\mathcal{S}_{b,k} = \{s \in \mathcal{S} | \delta(s) \leq k\}$, where \mathcal{S} is the set of states in the model and $\delta(s) - 1$ is the least number of intermediate states between state s and the (nearest) initial state s_0 (or with other words, $\delta(s)$ is the minimal distance from s_0).

Within a given bound k , it is not sure that the given formula f can be evaluated. For instance, if $\text{EF } p$ is false for a given bound k , it means that there are no states within the given bound that satisfies p . But we cannot answer, if there is such state outside bound k or not. Therefore the bounded model checking has to be continued with a greater k parameter until it will be true or the full state space will be explored, i.e. $\mathcal{S}_{b,k} = \mathcal{S}$. This iterative workflow can be seen in Figure 1.1

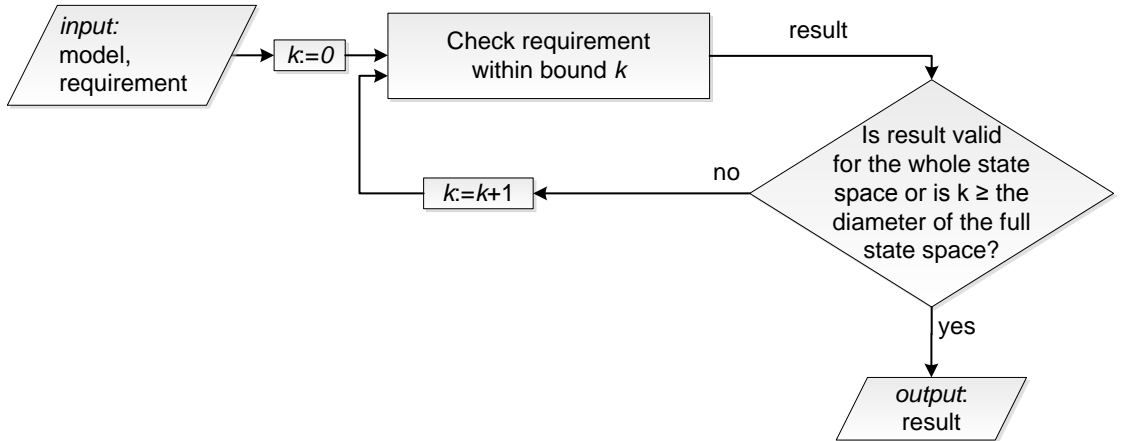


Figure 1.1: Workflow of bounded model checking

1.1.2 Symbolic model checking

First model checking methods handled and stored each state individually. These are the so-called *explicit model checking* methods. The main problem is their scalability: even with modern computers, it is impossible to store large number ($> 10^{10}$) of states in the memory.

In 1987, Ken McMillan “realized that by using a symbolic representation for the state transition graphs, much larger systems could be verified” [18]. It led to the birth of *symbolic model checking*. With symbolic techniques, it is possible to manipulate entire set of states together, not just individual states.

First attempts used Binary Decision Diagrams (BDDs, cf. Section 1.3) to encode the state space. With the help of decision diagrams, state spaces with the size of 10^{20} states have become analysable by model checking [10]. (At that time, the explicit model checking methods were not able to deal with state spaces containing more than 10^6 reachable states.) As the BDD represents the state space symbolically, it can be much more compact than the explicit representation.

Since then, multiple new approaches are invented. One of them is the saturation-based algorithm, that will be discussed later in detail, in Chapter 2.

1.1.3 Computation Tree Logic

With atomic state propositions and Boolean operators, various properties of static models can be expressed. But if the model is dynamic (e.g., in a Petri net, transitions can be fired which modifies the current state of the model), in most cases the logical time (the order of events) has to be expressed too. Using a *temporal logic* formalism, we can express and examine the truth of logical expressions and their change during the time.

There are multiple existing temporal logic formalisms. Here, I will focus on the *Computation Tree Logic* (CTL) formalism. CTL was first introduced by Clarke and Emerson in [27]. Later, it was influenced by the syntax of UB (Unified Branching Time logic, presented in [5]). The final version of the CTL formalism can be found in [19].

CTL is a *branching time* logic, and it is interpreted over a computation tree which comes from the branching semantics of the system. The definition of the computation tree is as follows (based on [19]):

Definition 3 (Computation tree). Let M be a model with the set of states \mathcal{S} and let \mathcal{N} be the set of possible state-state transitions in the model. A *state path* is a sequence of states (s_0, s_1, \dots) such that $\forall i : (s_i, s_{i+1}) \in \mathcal{N}$. For any state $s_0 \in \mathcal{S}$, a *computation tree* is a tree rooted at s_0 , where the nodes are representing states, and $s \rightarrow t$ is an arc in the tree iff $(s, t) \in \mathcal{N}$. (Because computation tree is a tree, no cycles are allowed, therefore multiple nodes can represent the same state. Moreover, the computation tree is often infinite.) ▪

Informally, the computation tree with root s_0 is the most compact tree that has the same paths from root as the possible state paths starting from state s_0 in the model.

The temporal expressions interpreted on the computation tree are the *CTL formulas* (or state formulas). The formal definition of a CTL formula is the following (based on the definition in [20]):

Definition 4 (CTL formula). A CTL formula (*state formula*) can be defined by the following rules:

- Every P atomic proposition is a state formula.
- If p and q are state formulas, then $\neg p$, $p \vee q$, and $p \wedge q$ are state formulas too.
- If p and q are state formulas, then $X p$, $F p$, $G p$, and $p U q$ are path formulas.
- If s is path formula, then $E s$ and $A s$ are state formulas. ▪

The definition gives us the syntax of the CTL formulas, but the semantics need to be defined too.

Semantics of CTL formula. The given definition permits eight possible operator pairs: EX, EF, EU, EG, AX, AF, AU, AG. These are the “building stones” of the CTL formulas. The intuitive semantics of these operator pairs are the following:

- EF p : p is true for at least one state on some (at least one) path,
- EG p : p is true for all states on some path,
- EX p : p is true for some next state,
- E[$p U q$]: p is true for a state on some path and for all intermediate states on those path q is true,
- AF p : p is true for at least one state on all path,
- AG p : p is true for all states on all path,
- AX p : p is true for all next state,
- A[$p U q$]: p is true for a state on all path and for all intermediate states in all path q is true.

Example 1. For example, $M, s_0 \models EX p$ is true (formally: $s_0 \in \{M, s_0 \models EX p\}$), iff there is at least one successor state s_1 of s_0 in M such that $s_1 \models p$. Similarly, $M, s_0 \models AX p$ is true, iff for every successor states s_1, s_2, \dots of s_0 holds: $s_1, s_2, \dots \models p$.

The semantics can be intuitively visualized on computation trees as it can be seen in Figure 1.2. (Every subfigure shows a computation tree where the given CTL formula is true.) The circles represent nodes in computation tree (which represent states). Black circles are states where property p holds, grey circles are states where property q holds.

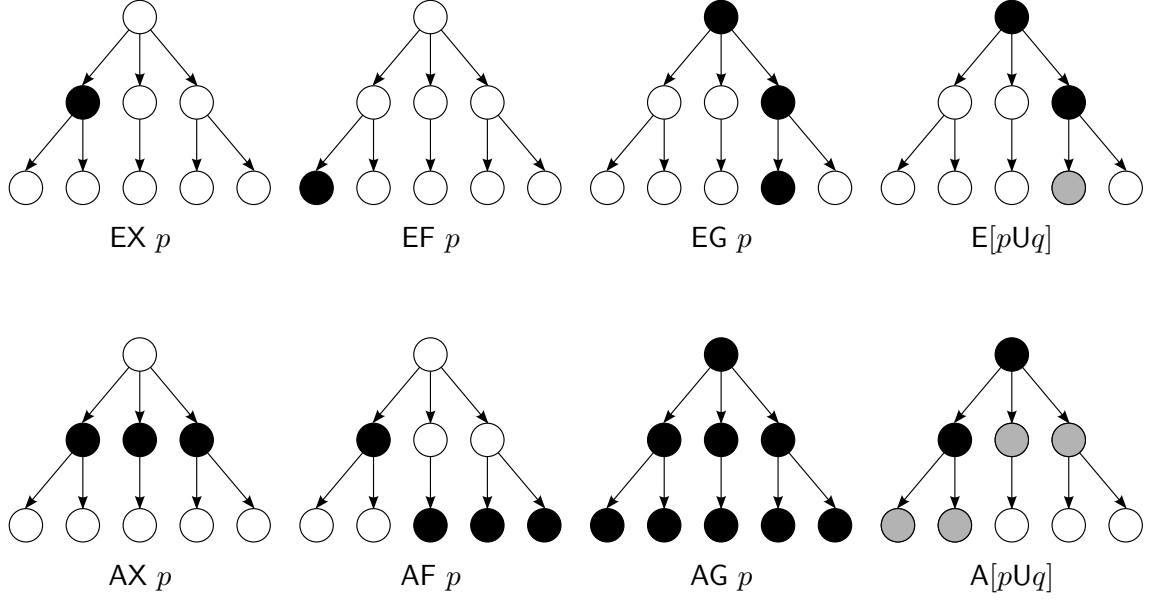


Figure 1.2: Examples for CTL operators

For my current work, the informal definition of the CTL semantics is enough. Formal definitions of Computation Tree Logic can be read for example in [16, 2].

The eight possible operator pairs are not independent from each other. All eight operator pairs can be expressed from a well chosen set of three operator pairs. For instance, we can express all operators using $\{EX, EU, EG\}$ in the following way [15]:

- $AX\ p \equiv \neg EX\ \neg p$,
- $AG\ p \equiv \neg EF\ \neg p$,
- $AF\ p \equiv \neg EG\ \neg p$,
- $A[p\ U\ q] \equiv \neg E[\neg q\ U\ (\neg p \wedge \neg q)] \wedge \neg EG\ \neg q$,
- $EF\ p \equiv E[true\ U\ p]$.

1.2 Petri nets

Petri net is a common modelling tool for system modelling and system analysis. Petri nets have both graphical and mathematical representation, so they are useful for visualization and analysis too.

Definition 5 (Petri net). An ordinary Petri net is a 5-tuple $PN = (P, T, E, w, M_0)$, where:

- $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places,
- $T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions ($P \cap T = \emptyset$),

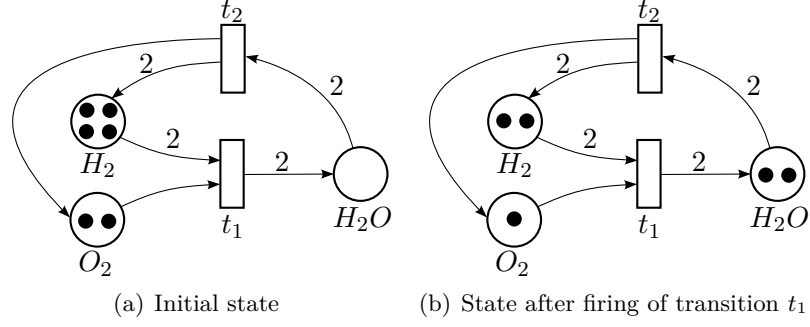


Figure 1.3: *Example Petri net*

- $E \subseteq (P \times T) \cup (T \times P)$ is a finite set of edges,
- $w: E \rightarrow \mathbb{Z}^+$ is the weight function assigning weights to edges,
- $M_0: P \rightarrow \mathbb{N}$ is the initial token distribution (initial marking) [41].

Dynamic behaviour. The *dynamic behaviour* of the Petri net is determined by the *firing* of transitions. The rules of firing are the following [41]:

- A transition $t \in T$ is *enabled*, iff $\forall p \in \{p' | \exists (p', t) \in E\} : M(p) \geq w(p, t)$ (where $M(p)$ is the current marking of place p). In other words: a transition $t \in T$ is enabled, iff for every input place there are at least as many tokens as the weight of the corresponding incoming edge.
- The firing of an enabled transition t is nondeterministic.
- If the enabled transition t *fires*, it decreases the number of tokens for every incoming place p' by $w(p', t)$ and increases the number of tokens for every outgoing place p'' by $w(t, p'')$.

Graphical representation. Graphically, a Petri net is a directed, weighted bigraph, where the two vertex sets are T (transitions) and P (places). Transitions are represented by rectangles, places are represented by circles. The tokens are shown as dots inside the places.

Example 2. Figure 1.3 shows a Petri net that models a simple chemical process [41]. There are two transitions (t_1 and t_2) and three places (H_2 , O_2 , and H_2O) in the Petri net. In Figure 1.3(a), only the transition t_1 is enabled. Firing the transition t_1 changes the state of the net to the state in Figure 1.3(b). In this new state, both transitions t_1 and t_2 are enabled.

Extension with inhibitor edges. The ordinary Petri nets can be extended with *inhibitor edges*. An inhibitor edge is a special edge in a Petri net which can disable the firing of a transition.

Definition 6 (Petri net extended with inhibitor edges). The Petri net extended with inhibitor edges is a tuple $PN_I = (PN, I, w_I)$, where:

- PN is an ordinary Petri net,
- $I \subseteq (P \times T)$ is a finite set of inhibitor edges,
- $w_I: I \rightarrow \mathbb{Z}^+$ is the weight function assigning weights to inhibitor edges. .

The firing rule is also modified: $\forall (p, t) \in I$: if $M(p) \geq w_I(p, t)$ then transition t is not enabled. This extends the expressive power of Petri nets, but the analysis of these nets can be much more difficult, therefore some analysis methods does not apply for these models [11].

1.3 Decision diagrams

The usage of multivariable functions is common in many fields, thus finding a way to represent them well is widely needed. The easiest representation method is the application of the *binary decision trees* providing a representation of Boolean functions. This representation is quite straightforward but it is not compact enough for storing or representing larger functions.

1.3.1 Binary Decision Diagrams

In 1986, Randal Bryant introduced an efficient representation for the binary functions $f(x_1, \dots, x_n) \rightarrow \{0, 1\}$ (where $x_1, \dots, x_n \in \{0, 1\}$) [9] called binary decision diagram. Its definition is the following:

Definition 7 (Binary Decision Diagram). A *binary decision diagram* (BDD) is a directed acyclic graph (DAG). This graph has two types of nodes in the vertex set (V): terminal and nonterminal nodes. Every *nonterminal* node $v \in V$ has two outgoing edges pointing to two children nodes: to $v[0] = low(v) \in V$ and to $v[1] = high(v) \in V$. Also, every nonterminal v has a level number too: $level(v) \in \mathbb{Z}^+$. For every nonterminal v , $level(low(v)) < level(v)$ and $level(high(v)) < level(v)$. There are also exactly two *terminal* nodes, $\mathbf{0} = w_0 \in V$ and $\mathbf{1} = w_1 \in V$. Similarly to nonterminal nodes, the terminal nodes have level numbers: $level(w_0) = level(w_1) = 0$. The terminal nodes have binary *values* too: $value(w_0) = 0$, $value(w_1) = 1$. We call $w_0 = \mathbf{0}$ as *terminal zero* and $w_1 = \mathbf{1}$ as *terminal one*.

Every BDD has a *root node* which is on the highest level (top level). On this level, the root node is the only node. .

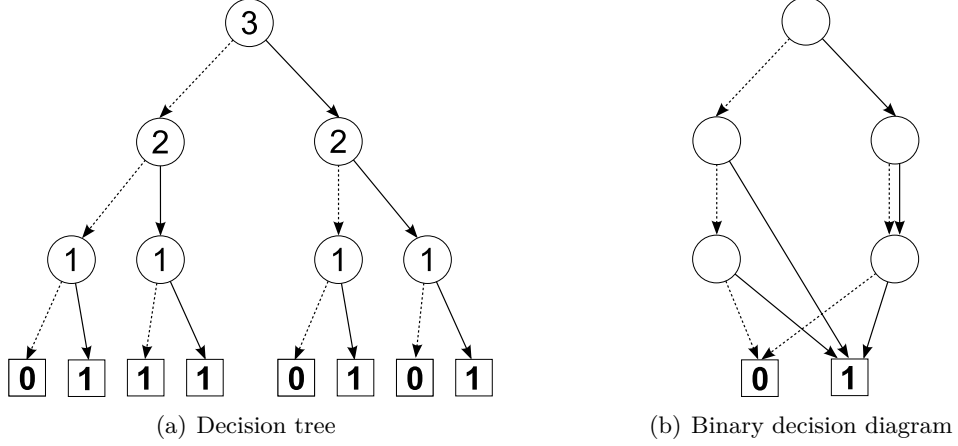


Figure 1.4: Encoding a binary function

The BDD formalism has several advantages: it is efficient (can be a compact representation), easy to handle and general (not tied particularly to one model checking method) [6].

Semantics of BDDs. A v -rooted BDD represents the following f_v function [9]:

- If node v is terminal, $f_v = \text{value}(v)$.
- If node v is nonterminal,

$$f_v(x_n, \dots, x_1) = \bar{x}_{\text{level}(v)} \cdot f_{\text{low}(v)}(x_n, \dots, x_1) + x_{\text{level}(v)} \cdot f_{\text{high}(v)}(x_n, \dots, x_1)$$

Graphical representation of BDDs. The nonterminal nodes of BDDs are represented by circles, and the terminal nodes with squares. The squares of terminal nodes contain the value of the terminal nodes. Edges from $v \in V$ to $\text{low}(v)$ are drawn with dotted lines, edges from $v \in V$ to $\text{high}(v)$ are drawn with solid lines.

Example 3. In Figure 1.4(b), we can see a binary decision diagram (BDD) encoding function $f(x_3, x_2, x_1)$ which is true (formally: which assigns 1) for the following 3-tuples: $(0, 0, 1)$, $(0, 1, 0)$, $(0, 1, 1)$, $(1, 0, 1)$, $(1, 1, 1)$. For instance, $f(0, 1, 1) = 1$, but $f(0, 0, 0) = 0$. (Note: this BDD is not in a canonical format, therefore it is not in its most compact form.)

The binary decision tree in Figure 1.4(a) encodes the same function f . In this case, it can be seen that the BDD representation is more compact than the decision tree.

1.3.2 Multivalued Decision Diagrams

The *multivalued (or multiway) decision diagram* (MDD) is an extension of the previously introduced BDD. In the case of MDDs, the variable x_i of the encoded $f(x_n, \dots, x_1) \rightarrow$

$\{0, 1\}$ function can have values from a finite domain D_i , not only from the binary domain $(\{0, 1\})$ as in a BDD. In other words, for every variable x_i , $x_i \in D_i = \{d_{i,0}, d_{i,1}, \dots, d_{i,|D_i|-1}\}$. Without loss of generality, we can use integer domains, thus for every x_i , I will use domains $D_i = \{0, 1, \dots, |D_i| - 1\}$.

Structure. In MDDs —analogous to BDDs—, there are terminal and nonterminal nodes. The terminal nodes are the same in MDDs: terminal zero and terminal one. Contrarily, the nonterminal nodes are different. Every nonterminal $v \in V$ has $|D_{level(v)}|$ outgoing edges pointing to nodes $v[0], \dots, v[|D_{level(v)}| - 1]$.

Obviously, every BDD is a special MDD, where $D_i = \{0, 1\}$ for every $i \in \{1, \dots, n\}$.

The semantics of MDDs are also similar to BDDs. I will discuss it in details after the introduction of canonical MDDs.

Graphical representation. As in the case of BDDs, we represent the nonterminal nodes of MDDs with circles, and the terminal nodes with squares. The squares of terminal nodes contain the value of the terminal nodes. The edge from $v \in V$ to $v[i]$ is drawn with solid line and labelled with the number i .

Canonical MDDs. For easier handling and representation, there are some canonical forms of MDDs.

- In a *canonical MDD*, there are no duplicated nodes on the same level. If for $v, w \in V$, $level(v) = level(w)$ and $\forall i \in D_{level(v)}: v[i] = w[i]$, then $v = w$.
- A *quasi-reduced MDD (QMDD)* is a canonical MDD, where is no level skipping. In other words, $level(v) - 1 = level(v[i])$ for every $i \in D_{level(v)}$ and for every nonterminal node v .
- In a *fully-reduced MDD*, there are no duplicated nodes (but level skipping is possible, thus it is possible to be true for some v and i : $level(v[i]) < level(v) - 1$). If $\forall i \in D_{level(v)} \cup D_{level(w)}: v[i] = w[i]$, then $v = w$.

The definition of the semantics of the general MDDs would be complex. As the algorithms in this thesis only use QMDDs, only the semantics of this subclass will be defined here.

Semantics of QMDDs. First —for shorter representation— let $v[i_k, i_{k-1}, \dots, i_1]$ denote $(\dots((v[i_k])[i_{k-1}]) \dots)[i_1] = v[i_k][i_{k-1}] \dots [i_1]$.

The v -rooted quasi-reduced MDD representing function f_v has the following meaning:

$$f_v(x_n, \dots, x_1) \big|_{x_n=i_n, \dots, x_1=i_1} = f_v(i_n, \dots, i_1) = 1 \Leftrightarrow value(v[i_n, \dots, i_1]) = 1$$

The *intersection* of nodes v and w (taking place on the same level: $level(v) = level(w)$) is the following:

$$v \cap w = \begin{cases} v \wedge w & \text{if } level(v) = level(w) = 0 \\ z & \text{else, where } z[i] = v[i] \cap w[i] \text{ for all } i \in D_{level(v)} \end{cases}$$

Similarly to the common Boolean logic, if v and w are terminal nodes, $v \wedge w = \mathbf{1} \Leftrightarrow value(v) = value(w) = 1$.

The *union/intersection of two QMDDs* is the same as the union/intersection of their root nodes.

1.3.3 Edge-valued Decision Diagrams

This section is based on [14]. The *edge-valued decision diagrams* (EDDs) are extended MDDs, introduced in [14]. (Note: the authors of [14] call this decision diagram type as EV^+MDD . I will use the simpler name *EDD* instead.) With the help of the EDDs, functions $f(x_n, \dots, x_1) \rightarrow \mathbb{N} \cup \{\infty\}$ can be represented. For that reason, we assign *labels* to every edge in the EDD. Therefore the i th children of a node v is a $v[i] = \langle s, w \rangle$ tuple, where w is a node, and $s \in \mathbb{N} \cup \{\infty\}$ is a label assigned to the corresponding edge.

Definition 8 (Edge-valued decision diagram). An EDD is a (V, E) directed graph, where:

- $V = V' \cup \{\perp\}$, where \perp is the only terminal node, every other $v \in V'$ nodes are nonterminal nodes. ($\perp \notin V'$)
- Every node v has a level number denoted by $level(v)$. The level of the terminal node is zero: $level(\perp) = 0$. For all $v \in V' : level(v) > 0$.
- Every nonterminal node v on level l has $|D_l|$ outgoing edges, pointing to $v[0].node, \dots, v[l-1].node$.
- Every edge has a label $s \in \mathbb{N} \cup \{\infty\}$. The label of the i th edge of node v is denoted by $v[i].label$.
- The EDD has a root node r on the top level. On this level, there are no other nodes.
- The root node r has a “virtual” ingoing edge with a label ρ assigned to it. This ρ value is the so-called *dangling edge weight*. [14] ▪

Note: if $v[i].node = w$ and $v[i].label = s$, we can also denote it as $v[i] = \langle s, w \rangle$.

Definition 9 (Canonical EDD). An EDD is *canonical* if for every $v \in V'$ there is at least one edge labelled with zero [14]. ▪

Definition 10 (Quasi-reduced EDD). An EDD is *quasi-reduced* if it is canonical, there are no duplicates on a level (If $level(v) = level(w)$ and $\forall i \in D_{level(v)}: v[i] = w[i]$, then $v = w$), and for every $v \in V'$ every outgoing edge points to node on level $level(v) - 1$ [14] (there is no level skipping). \blacksquare

Semantics of EDDs. The r -rooted EDD represents the following $f(x_n, \dots, x_1)$ function:

$$\begin{aligned} f(i_n, \dots, i_1) = s &\Leftrightarrow r[i_n] = \langle s_{n-1}, v_{n-1} \rangle, \\ v_{n-1}[i_{n-1}] &= \langle s_{n-2}, v_{n-2} \rangle, \\ &\dots, \\ v_1[i_1] &= \langle s_1, \perp \rangle, \\ s &= s_1 + \dots + s_{n-1} + \rho \end{aligned}$$

Graphical representation. A node of an EDD is often visualized as a rectangle with k slots, if the node has k outgoing edges. For each node, the i th outgoing edge starts from the i th slot of the node. The label of this i th edge is shown in the i th slot of the node. The zero-valued dangling edges and the edges with ∞ labels are omitted.

Example 5. The Figure 1.6 shows an EDD encoding the function $f(x_2, x_1)$. For this function, $f(0, 0) = 0 + 0 = 0$, $f(0, 1) = 0 + 3 = 3$, $f(1, 0) = 1 + 1 = 2$, etc.

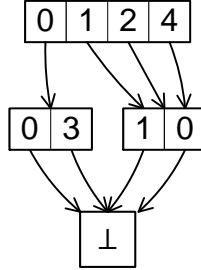


Figure 1.6: Edge-valued decision diagram

Chapter 2

Saturation

This chapter introduces the basic idea of the saturation techniques and the former saturation-based algorithms. First, a brief overview of the saturation techniques can be found in Section 2.1. The common base of all saturation-based algorithms is discussed in Section 2.2. Section 2.3 is dedicated to the challenges common for all the saturation algorithms. Then the main, specific saturation-based algorithms are introduced: the unbounded state space exploration algorithms (Section 2.4), the unbounded CTL model checking algorithms (Section 2.5), the bounded state space exploration algorithms (Section 2.6), and the bounded CTL model checking algorithms (Section 2.7). It has to be noted that we implemented all presented algorithms in our PetriDotNet analysis framework [55]¹.

2.1 Overview of the saturation-based algorithms

This section provides a general overview of the saturation algorithm (its definition, data structures and advantages). There are multiple saturation algorithms for different purposes, they will be introduced later in this chapter.

Saturation is a symbolic model checking method introduced in [12] by G. Ciardo et al. The former symbolic methods were typically efficient for synchronous systems. The aim of saturation is to provide a symbolic model checking solution which works well for asynchronous systems.

This algorithm can be efficient thanks to its special iteration strategy and to the applied data structures that decrease the small peak memory consumption and it represents the state space in a compact form.

¹All the presented algorithms are implemented by Attila Jámbor and myself.

2.1.1 Definitions of model and states

Saturation works on an abstract discrete-state model, defined as follows. (The following definitions are from [15, 12, 54, 47].)

Definition 11 (Discrete-state model). A discrete-state model is a $(\hat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{E}, \mathcal{N})$ tuple, where:

- $\hat{\mathcal{S}}$ is the potential state space,
- $\mathcal{S}^{init} \subseteq \hat{\mathcal{S}}$ is the set of initial states (usually $|\mathcal{S}^{init}| = 1$ for Petri nets),
- \mathcal{E} is the set of events,
- $\mathcal{N}: \hat{\mathcal{S}} \rightarrow 2^{\hat{\mathcal{S}}}$ is the next-state function, describing which states can be reached from a given state in a single step. ▪

Dynamic behaviour of the model. A model has a *current state* \mathbf{s} in every moment. Initially, $\mathbf{s} \in \mathcal{S}^{init}$. The current state can be modified by the *firing* of events. Next-state function \mathcal{N} describes which state-state transitions are allowed in the model.

Decomposition. The saturation algorithm works on a decomposed model. The model must be decomposed into K submodels. Each submodel has *local states* encoded by integers. The *local state space* of the i th submodel is marked with \mathcal{S}_i . Therefore, the potential state space of the model is $\hat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$. A *global state* in the discrete-state model is a K -tuple of local states, i. e., $\mathbf{i} = (i_K, \dots, i_1)$.

The next-state function can be decomposed to: $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$, where \mathcal{N}_α is the next-state function of the event α . This is common in the analysis of asynchronous systems [13].

Reachable state space. The (*reachable*) *state space* \mathcal{S} of a model is the set of reachable states from the initial states. Formally: $\mathcal{S} = \mathcal{S}^{init} \cup \mathcal{N}(\mathcal{S}^{init}) \cup \mathcal{N}(\mathcal{N}(\mathcal{S}^{init})) \cup \dots = \mathcal{N}^*(\mathcal{S}^{init})$. (Note that $\mathcal{S} \subseteq \hat{\mathcal{S}}$, because $\hat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$ and for every global state $\mathbf{i} = (i_K, \dots, i_1) \in \mathcal{S}$, $\forall k : i_k \in \mathcal{S}_k$.)

2.1.2 Data structures

It is important to store the parts of the discrete-state model efficiently. Nearly all variants of the saturation method use decision diagrams to store the state space \mathcal{S} and the next-state function \mathcal{N} . This section shows, how the different necessary elements encoded and stored.

Encoding state spaces. As the state space is a set of global states, and the global states are K -tuples, it is possible to use Multivalued Decision Diagrams (MDDs) to store the state space.

To store the state space of a model decomposed into K submodels, we need an MDD with $K + 1$ levels (K nonterminal levels and 1 terminal level). For the sake of simplicity, the saturation algorithms use quasi-reduced MDDs to store the state space. Each level in the MDD is assigned to a submodel. If the i th submodel has local state space \mathcal{S}_i , the MDD nodes on level i have $|\mathcal{S}_i|$ outgoing edges towards level $i - 1$ (i.e., $|D_i| = |\mathcal{S}_i|$).

After encoding, the (reachable) state space \mathcal{S} contains the global state $\mathbf{i} = (i_K, \dots, i_1)$ if and only if for the v -rooted MDD representing \mathcal{S} : $value(v[i_K, \dots, i_1]) = 1$.

Example 6. *This example shows the state space of the model in Figure 1.3 encoded by decision diagram. First, Figure 2.1(a) shows the partitioning of the model: the first submodel contains places H_2 and O_2 , the second submodel contains the place H_2O .*

Figure 2.1(b) and 2.1(c) shows the local states of the submodels. In the tables, i_j represents a local state of the j th submodel. The order of the local states is not important, the only restriction is that the initial state is always the local state 0.

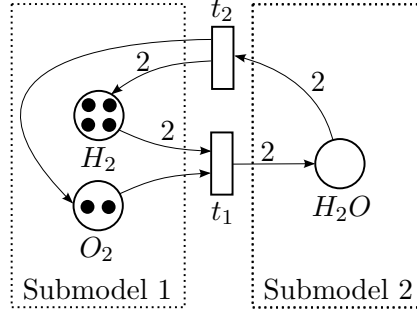
After the partitioning and encoding of local states, the set of reachable states (\mathcal{S}) can be expressed by an MDD as it can be seen in Figure 2.1(d). The following global states (i_2, i_1) are reachable: $\{(0, 0), (1, 2), (2, 1)\}$.

Encoding next-state functions. As can be read earlier, the next-state function is a function $\mathcal{N}: \hat{\mathcal{S}} \rightarrow 2^{\hat{\mathcal{S}}}$. But also it can be interpreted as a set too: $\mathbf{j} \in \mathcal{N}(\mathbf{i}) \Leftrightarrow (\mathbf{i}, \mathbf{j}) \in \mathcal{N}$. Therefore, the next-state function can also be stored as an MDD. Because the items of this “set” contains pairs of K -tuples, the representing MDD has to have $2K + 1$ levels ($2K$ nonterminal and 1 terminal level).

To be able to exploit the locality of events, it is common to assign MDD levels to submodels in an *interleaved* way. Let be the levels of the MDD numbered with $K, K', \dots, 1, 1', 0$, where $K, \dots, 1$ levels correspond to “from” states and $K', \dots, 1'$ correspond to “to” states. In this way, for the v -rooted MDD encoding \mathcal{N} : $(\mathbf{i}, \mathbf{j}) \in \mathcal{N} \Leftrightarrow value(v[i_K, j_K, \dots, i_1, j_1]) = 1$ (assuming that $\mathbf{i} = (i_K, \dots, i_1), \mathbf{j} = (j_K, \dots, j_1)$).

The next-state function can be split into multiple partitions to optimize the storage. For instance, we can store the next-state functions of each event separately (\mathcal{N}_α for each α event). In this way, the global function \mathcal{N} can be produced using the union MDD operation: $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$. (These are the *disjunctive partitions* of the function.)

Note: some saturation methods use matrices (so-called Kronecker-matrices) instead of decision diagrams to store the next-state function [13]. In this thesis, I will only use decision diagrams to encode next-state functions, as our former experiments showed better results with MDDs, furthermore it is a more general way to represent the next-state functions [25].



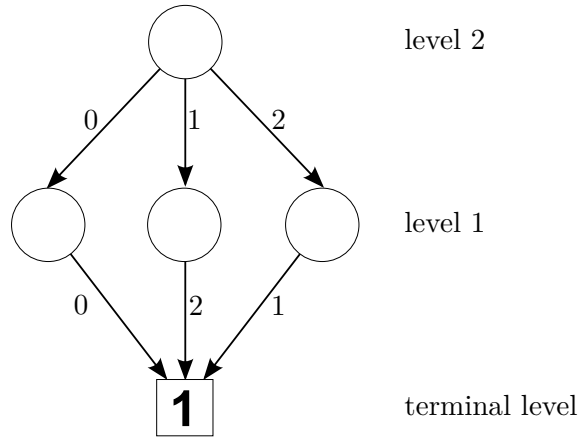
(a) The partitioned model

i_1	$M(H_2)$	$M(O_2)$
0	4	2
1	2	1
2	0	0

(b) Local state encoding in submodel 1

i_2	$M(H_2O)$
0	0
1	4
2	2

(c) Local state encoding in submodel 2



(d) Set of reachable states, encoded by MDD

Figure 2.1: State space encoding example

Relational product. If the next-state function is encoded by an MDD, it is useful to introduce a new operator: the relational product operator. Formally this operator is the following [16]:

Definition 12 (Relational product operator). Let \mathcal{X} be a set of (i, j) pairs and let \mathcal{Y} be a set. $RelProd(\mathcal{X}, \mathcal{Y}) = \{j : (\exists i)(i, j) \in \mathcal{X} \wedge i \in \mathcal{Y}\}$. ▪

This operator is used to compute the successor states of a given set of states. Let \mathcal{S}' be a set of states encoded by an MDD (with K nonterminal levels) and \mathcal{N} the next-state function encoded by an MDD with $2K$ nonterminal levels. The successors of states in \mathcal{S}' are $RelProd(\mathcal{N}, \mathcal{S}')$.

Events. Events are the concepts of the higher level modelling formalisms and we use them to exploit locality.

Definition 13 (Independent event). An event α is *independent* of the submodel k , if for every state $\mathbf{i} = (i_K, \dots, i_1)$ the enabling of α does not depend on i_k and its firing will not modify i_k . Formally: $\forall \mathbf{i} = (i_K, \dots, i_1) : \exists \mathbf{j} = (j_K, \dots, j_1) \in \mathcal{N}_\alpha(\mathbf{i}) \Rightarrow j_k = i_k$. ▪

The submodel k belongs to the support set of event α (denoted by $supp(\alpha)$), if α is not independent of submodel k . $Top(\alpha) = \max supp(\alpha)$ denotes the maximal numbered submodel in $supp(\alpha)$ and $Bot(\alpha) = \min supp(\alpha)$ denotes the minimal numbered submodel in $supp(\alpha)$ (i.e., $Top(\alpha)$ is the highest and $Bot(\alpha)$ is the lowest level on which the event α depends). Evidently, $1 \leq Bot(\alpha) \leq Top(\alpha) \leq K$. In the following, \mathcal{E}_k depicts the events with $Top = k$ (formally, $\mathcal{E}_k = \{\alpha \in \mathcal{E} | Top(\alpha) = k\}$).

Connection between Petri nets and global states. The introduced discrete-state model is a high-level view of a model. Petri nets can be regarded as discrete-state models too. Let P be the set of places in a Petri net. This set can be partitioned into K disjoint subsets: P_1, \dots, P_K . Each set P_i with the connected edges and transitions represent a submodel in the discrete-state model. The local states spaces (\mathcal{S}_i) are the possible combination of markings, encoded by integers.

Note that the contents and the ordering of the P_i subsets can have a huge effect on the run time and memory consumption of the saturation algorithm.

The transitions of a Petri net can be regarded as the events of the discrete-state model ($\mathcal{E} = T$).

The initial state (\mathcal{S}^{init}) is the global state of the net with the initial marking assigned. For Petri nets, $|\mathcal{S}^{init}| = 1$ as there is only one single initial state.

Regarding this, it is easy to see that saturation-based methods working on discrete-state model can be applied to Petri nets. Also, it justifies, why Petri nets can be used as models for model checking.

2.1.3 Advantages of saturation

In this section, the main advantages of saturation are introduced.

- *Symbolic technique* As saturation is a symbolic technique, it benefits from the advantages of symbolic techniques: the compact state space representation and the ability to operate on large set of states together.
- *Minimizing peak memory consumption* Generally, the symbolic methods can store the state space in a compact form, but usually the peak memory consumption during the exploration is higher than the final state space. The saturation iteration strategy and its partitioned next-state function help to keep the peak memory consumption low [13].
- *Not required to know possible local state spaces (\mathcal{S}_k) a priori* One of the strength of the saturation algorithm is that it does not need to know the local state spaces a priori. Instead, it uses an *on-the-fly local state space exploration*, which extend the applicability of the algorithm [13]. (This property of the saturation will be discussed in details later.)

2.2 The background of saturation

As it was stated before, the saturation-based algorithms use decision diagrams to encode the set of explored global states of the model. Therefore a node with its subgraph in the state space decision diagram represents a set of states of some submodels. (The set of states represented by node v and its subgraph is denoted by $\mathcal{B}(v)$, as stated earlier.)

The most important definition related to the saturation-based algorithms is the definition of the *saturated node*:

Definition 14 (Saturated node). A decision diagram node v is *saturated* “if it encodes a set of states that is a fixed point with respect to the firing of any event at its level or at a lower level”, i. e., if $\mathcal{B}(v) = \mathcal{N}_{\leq \text{level}(v)}^*(v)$ holds [12]. ▪

There are two important consequences of this definition. First, if v is saturated, all node in its subgraph have to be saturated too. Second, if the root node r of the decision diagram is saturated, the diagram encodes $\mathcal{B}(r) = \mathcal{N}^*(r)$, therefore it encodes a fixed point of the initial state(s), which is equivalent to the reachable state space $\mathcal{S} = \mathcal{B}(r)$.

Therefore, the goal of saturation is to saturate the root node of the state space, and for that reason, the lower nodes have to be saturated too recursively.

Informal iteration order. Informally, the following tasks have to be done in order to explore the state space [16]. (Reminder: \mathcal{E}_i means the set of events e such that $Top(e) = i$, i. e., $\mathcal{E}_i = \{e \in \mathcal{E} \mid Top(e) = i\}$.)

- Build the decision diagram encoding of initial state(s) \mathcal{S}^{init} of the model.
- Saturate nodes on level 1: fire all events in \mathcal{E}_1 on them.
- Saturate nodes on level 2: fire all events in \mathcal{E}_2 on them. If it creates new nodes on level 1, saturate them immediately.
- ...
- Saturate nodes on level k : fire all events in \mathcal{E}_k on them. If it creates new nodes on level i ($1 \leq i < k$), saturate them immediately.
- ...
- Saturate the root node on level K : fire all events in \mathcal{E}_K on it. If it creates new nodes on level i ($1 \leq i < K$), saturate them immediately.

This iteration order is implemented by multiple functions. Basically, all saturation-like algorithm consists of four functions (methods), as follows (based on [13]).

SatRecFire This method fires a given event α on a given node p .

First, it computes the possible successor states of $\mathcal{B}(p)$ reachable in one step (by calling SatRecFire recursively). Then, it saturates the built node. So basically this method performs the $RelProd(\mathcal{B}(n_\alpha), \mathcal{B}(p))$ operation (where n_α is the node of the \mathcal{N}_α relation corresponding to p) and saturates every new node.

After that, the result of this method will be a *new decision diagram subgraph* that encodes the following: $\mathcal{N}_{\leq level(p)}^*(\mathcal{N}_{\alpha, \leq level(p)}(\mathcal{B}(p)))$ which is a fixed point of the possible successor states.

SatFire This method fires a given event α *exhaustively* on a given node p . It fires α on all lower nodes by using SatRecFire. The SatFire function uses a method called “chaining”: if the firing of α resulted a local state j from that α is still fireable, it will be fired too using SatRecFire.

As a result of SatFire, node p will be *updated* to encode $\mathcal{N}_{\leq level(p)-1}^*(\mathcal{N}_\alpha^*(\mathcal{B}(p)))$, in other words, the modified p will encode all states reachable from the given p by firing α .

It is important to note that SatFire can find new reachable states by firing the same event α again after invocation SatFire with other events on the same node p . Therefore multiple SatFire calls might be needed to explore all reachable states with the same event α on the same node p . (The function Saturate takes care of it.)

Saturate The Saturate method saturates a node p by firing *all events* e such that $Top(e) = level(p)$. (It is not necessary to fire the events with $Top(e) < level(p)$, because they cannot affect p .) If firing of any event e resulted a new state, all events $e' \in \mathcal{E}_{level(p)}$ will be fired once again, because other events can be enabled in the new state (from the updated node p).

After Saturate method, the given node p will be saturated. In other words, this method computes $\mathcal{N}_{\leq level(p)}^*(\mathcal{B}(p))$.

Generate The Generate method builds the decision diagram encoding the initial state (assuming that $\mathcal{S}^{init} = \{s_0\}$) and saturates the nodes of initial state in bottom-up order.

The result of Generate method will be a decision diagram that encodes the reachable state space \mathcal{S} , which is the goal of state space exploration.

2.3 Challenges

The basic idea of saturation was introduced in Section 2.2, but there are some further challenges:

1. The decision diagram encoding the state space must be kept in a quasi-reduced form.
2. As the algorithm recursively calls SatRecFire methods on nodes of a decision diagram, the method can be called on the same node several times. As the result will be the same each time, this calculation is unnecessary. Furthermore, it can introduce exponential number of redundant method calls.
3. This introduction assumed that the possible local states and the next-state functions \mathcal{N}_α of each event α are known a priori. However, in most cases it is not true. For example, in a Petri net, the maximal amount of tokens that can be contained by a specific place is not known trivially from the model itself without state space exploration.

The solutions given to these problems can be read in the following. Note that the solutions are not my own work, they were introduced by Ciardo et al. [12, 13] as they are necessary to implement every saturation-based method.

Keeping the decision diagram in quasi-reduced form. The first problem can be solved easily. To ensure the quasi-reduced form, before inserting a node v to the level k , the uniqueness of v must be checked. For that reason, for every level k , the set of nodes (“unique table” of nodes) are maintained. If there is a node v' already inserted to the unique table of this level, the node v will be substituted by v' and it will not be inserted again.

Thus a method called **CheckIn**(v) is used to insert node v to the decision diagram. If there is already a node v' in the unique table such that $\mathcal{B}(v) = \mathcal{B}(v')$, the **CheckIn** method returns v' . Otherwise, it inserts v to the unique table and returns the same v . [12, 13]

There is another problem connected to the quasi-reduced diagrams. If a node z encodes empty set ($\mathcal{B}(z) = \emptyset$) in a fully reduced decision diagram, it must be substituted by terminal node $\mathbf{0}$. Therefore, it is easy to decide whether a subgraph of a node encodes any state or not. Contrarily, in a quasi-reduced diagram, “level skipping” is not allowed, so this substitution is not possible. For that reason, nodes are added to encode the empty set for every level during the initialization of the decision diagram. These nodes have a flag “zero node”, so it can be determined easily for a node, if it encodes any states or not: it encodes some states iff it is not a “zero node” (because two identical nodes are not allowed in quasi-reduced diagrams). In the pseudocodes, all zero nodes are marked with the same symbol $\mathbf{0}$ as the terminal zero, as their semantics are the same and there is exactly one “zero node” on each level.

Omitting unnecessary calls. To eliminate unnecessary **SatRecFire** calls, a “fire cache” is maintained. As the **SatRecFire** method does not modify the given node (nor its subgraph), two **SatRecFire** method on the same node will produce the same subgraph as a result. Therefore the results of this method can be cached. This cache is checked at the beginning of the **SatRecFire** method and if the current calculation is already done, the cached value will be returned. Otherwise, it executes the calculation and puts the result into the cache. [12, 13]

A similar problem occurs in the case of decision diagram operations. As the result of union (or intersection) is the same for the same nodes, it can be cached too (in “union cache” and “intersection cache”).

Caching is necessary for saturation-based methods, because it eliminates exponential number of unnecessary calls.

On-the-fly local state space exploration. Typically, only the initial states are known before state space exploration, thus other local states cannot be encoded symbolically. Furthermore, only a part of next-state relation can be built: the transitions from initial states to other states (i.e., the states reachable with one transition firing from the initial state s_0 : $\mathcal{N}(s_0)$).

How can we find new local states? We can check on the model (for example on a Petri net), which new *local states* are reachable from the already known local states (without regarding to other local states). These new local states can be added to the set of possible states $\hat{\mathcal{S}}_k$ in level k . However, it is not sure that these local states can be reached *globally*, so we should not search local states from them. We call those states *unconfirmed*.

If it is found out that a previously unconfirmed local state i on level k is reachable globally (not just locally), we mark it *confirmed*. It consist of the following operations:

- Local state i will be added to the set of confirmed local states on level k (denoted by \mathcal{S}_k).
- New local states will be searched that are reachable from i by one single step. The newly explored states will be marked as unconfirmed.
- The next-state functions \mathcal{N}_α will be updated with possible transitions from i by firing α . (Note that because the next-state function will be updated only by transitions from confirmed states, the nodes on the “from” levels have outgoing edges labelled only with confirmed states, but on “to” levels, both confirmed and unconfirmed states are possible too as labels.)

The local states representing the initial state(s) are reachable trivially (without any firing), therefore they are confirmed during the initialization phase. (If there is only one initial state, the corresponding global state is usually encoded as $\mathbf{s}_0 = (0, 0, \dots, 0)$.)

This idea was introduced in details in [13]. However, the authors used Kronecker matrices instead of MDDs to store the next-state functions. In that case, the update of a next-state function is a local operation (it does not affect levels other than the level of the newly found local state). When we use MDDs to encode \mathcal{N}_α , we have to find and update all nodes on the actual level in the subgraph of the root node of \mathcal{N}_α . While it is a bigger effort to update \mathcal{N} encoded by MDDs, its usage eliminates some limitations of the Kronecker matrices² [3].

2.4 Unbounded state space generation

After the introduced base of the saturation-based algorithms, the specific algorithms can be discussed in details. In this section I overview two state space generation algorithms: the classic state space exploration algorithm and its extended version, the constrained state space exploration algorithm.

2.4.1 Classic method – the base algorithm

The “classic state space generation” method is the relatively straightforward implementation of the introduced principles. It was first presented by Ciardo et al. in [12], however the following pseudocodes are modified, to be consistent with the other discussed methods. Furthermore, the algorithm in [12] used a simpler next-state function representation (they assumed that the decomposition of the model is Kronecker consistent).

The pseudocodes of the classic method can be seen in Section C.1 (in the Appendix).

²For interested readers: to use Kronecker matrices, a so-called *Kronecker consistent* decomposition has to be used. In a Kronecker consistent decomposition $\mathcal{N}_\alpha = \mathcal{N}_{\alpha,K} \times \dots \mathcal{N}_{\alpha,1}$ holds (where $\mathcal{N}_{\alpha,l}$ stands for the part of \mathcal{N}_α which corresponds to level l)[13]. While it is true for all decomposition of ordinary Petri nets, this method is not applicable for other models, like coloured Petri nets. The usage of MDDs is more general and it can be directly applied to coloured Petri nets [25].

2.4.2 Constrained method

In [54], Y. Zhao and G. Ciardo proposed a new variant of the saturation algorithm: the *constrained saturation*. They used another MDD, the *constraint* MDD to encode the allowed set of possible state space. In other words, it encodes which global states are allowed to be visited. The constrained saturation cannot add global states to the state space that are not included in the constraint MDD.

This method is based on the following observation [54]. Let s be an MDD node encoding a state and let r be the node encoding the corresponding next-state function. Let $cons$ be the constraint corresponds to s .

$$\mathcal{B}(t) = RelProd(s, r) \cap \mathcal{B}(cons) \Leftrightarrow \forall i' : \mathcal{B}(t[i']) = \bigcup_{i \in S_{level(s)}} RelProd(s[i], r[i][i']) \cap \mathcal{B}(cons[i'])$$

Informally that means the saturation of a state set encoded by s is restricted to the given constraint $cons$ iff for every possible state transition the “to” state is restricted to the corresponding descendant of the $cons$ constraint.

For this reason, the constraint MDD is traversed along with the state space decision diagram. If the **SatRecFire** (or the **SatFire**) method is called with the state space node s and the constraint node c , and it recursively calls **SatRecFire** with parameter $s[i]$ in the case of a local state transition $i \rightarrow j$, then $c[j]$ will be given too as the constraint parameter. Also, if the firing of the given event in a **SatRecFire** method leads to local state j , but in the given constraint node c , $c[j] = \mathbf{0}$, then this firing will not be executed, as it is not allowed by the constraint.

Why is it useful? Well, the idea of constraints does not help the unbounded state space generation. In [54] it was used for model checking purposes (see in Section 2.5). The constrained saturation algorithm showed its efficiency in CTL model checking. In addition, it can be also a powerful method for bounded state space generation.

Although primarily it is not a state space exploration algorithm, it can be drawn up as a state space generation algorithm. If we do so, we can see that the classic method (described in Section 2.4.1) is a special case of the constrained saturation algorithm, where all possible path ends in the terminal node $\mathbf{1}$.

The pseudocodes of the constrained saturation can be found in Appendix C.2.

2.5 Unbounded CTL model checking

Saturation is a powerful method to explore state spaces which is the first phase of the model checking. In addition, saturation can be used in the second phase of the model checking: in the formula evaluation phase. This section briefly introduces, how saturation can be used to evaluate CTL expressions. However, in this thesis I have not improved

the formula evaluation algorithms, I used my earlier implementations (partly presented in [24]).

In [15], an attempt was presented to use saturation to evaluate CTL formulas. In that work, the evaluation of EF, EG and EU operators are discussed. (It is easy to evaluate EX operators, and with the set of {EX, EG, EU} operators, all other CTL operators can be expressed.)

The results of [15] are the following:

- The EX operator does not need saturation as it is just a step back from a set of states.
- A saturation-based method was proposed for EG operator, but according to the cited paper, it was more efficient than the “traditional” EG method only in special cases.
- The proposed saturation-based algorithm for EU operators is complicated. In addition, this algorithm just partly exploits the saturation and it uses large amount of MDD operations which are relatively expensive. (It explores more states than necessary and the unwanted states are eliminated using a costly intersection operation.)

The proposed algorithms showed that saturation can partly be used to evaluate CTL expressions, and generally it was more efficient than non-saturation-based symbolic methods [15].

The next milestone in saturation-based CTL model checking was the introduction of constrained saturation [54]. This method enables us to evaluate EU operators more efficiently, omitting a large number of MDD intersection operations, only by constrained saturation.

In the following, I introduce briefly the saturation-based evaluation of each operator. The exact pseudocodes can be found in the cited papers.

Evaluation of EF operator. It is not necessary to implement the EF operator as it can be expressed using EU, but the EF operator is often used and its implementation is rather straightforward.

To find the set $\{s \in \mathcal{S} \mid M, s \models \text{EF } R\}$, saturation can be used easily. The task is to find all the states reachable from a given set of states (R). It can be done with the same algorithm as the state space exploration, but with inverted next-state functions (\mathcal{N}^{-1}). The initial set of states of this backward saturation is R . Note that the on-the-fly local state space exploration is not needed in the case of the backward saturation, because the local state spaces are already explored in the state space generation phase.

Evaluation of EU operator. To find the set $\{s \in \mathcal{S} \mid M, s \models E[R \cup Q]\}$, we have to find all states from where a state $q \in Q$ is reachable, but only through states $r \in R$. For this purpose, the constrained saturation can be used with inverted next-state functions (\mathcal{N}^{-1}). The saturation have to be started with initial states Q . The constraint is the state set R .

Note: it is possible that $Q \not\subseteq R$, but it is not a problem, as constrained saturation only checks the newly added states against the constraint. The initial state set Q is not checked against the constraint.

Evaluation of EG operator. The EG operator differs from the EF and EU operators, because EG needs a least fixed point, while the others (and saturation generally) compute a greatest fixed point. Therefore the method proposed in [15] does not use saturation. Instead, the computation of the set $\{s \in \mathcal{S} \mid M, s \models EG P\}$ starts from the state set P , and it computes the least fixed point of $P \cap \mathcal{N}^{-1}(P)$.

Common properties. As the CTL formula evaluation algorithms run after state space exploration, the exploration of local state spaces and the building of next-state functions are already done.

Also it is a common need of all algorithms above to calculate the inverted next-state function \mathcal{N}^{-1} . Because inversion of \mathcal{N} means the exchange of “from” and “to” levels, it can be done without any MDD operations. In the introduced algorithms only the `LocalStateTransitionsToExplore(r)` method uses the next-state functions. It have to be modified to compute $\{(j, i) : r[i][j] \neq \mathbf{0}\}$ instead of $\{(i, j) : r[i][j] \neq \mathbf{0}\}$, but this modification is a simple swap of variables.

2.6 Bounded state space generation

Originally, saturation-based algorithms can efficiently explore the whole set of reachable states (\mathcal{S}). However, in some cases, not only the set of reachable states are interesting, but their *distance* from the initial state(s) too, i.e. the minimum number of transition firings to reach the state from an initial state. Shortly after the publication of the original saturation algorithm, G. Ciardo and R. Siminiceanu introduced an extension to the original saturation-based algorithm to be able to determine and store the distance information along with the explored states [14]. Their first objective was the generation of shortest path in the state space which can be easily done, if the distance information is known for all states. Thus the algorithms introduced in [14] do full (unbounded) state space exploration, but they store the distances of the states as well.

For that reason, they extended the former MDD-based data structures. The new data structure is the Edge-valued Decision Diagram (EDD). To adapt the former algorithms to EDDs, only a few modification was necessary:

- EDD edges (node-label pairs) have to be used instead of MDD nodes.
- The **Union** operation has to be adapted to EDDs (**UnionMinimum** operation) to be able to handle the weights on the edges.
- In **SatFire** method, the distance of the result of the firing has to be increased by 1.
- The saturated nodes have to be normalized, in order to ensure the canonicity of the EDD. It means that every node that has outgoing edges with finite weights needs to have at least one outgoing edge with weight 0.

The normalization of node p is quite simple, as it has to determine the minimal label m on the outgoing edges. Then for all outgoing edge, their label have to be decreased with m , and the ingoing edges of p have to be increased with m . (See the pseudocode of normalization on Algorithm C.17.)

The **UnionMinimum** operation is simple too. On the terminal node level, **UnionMinimum**($\langle v, \perp \rangle, \langle w, \perp \rangle$) returns $\langle \min(v, w), \perp \rangle$. Otherwise, it recursively applies **UnionMinimum** for all children nodes and normalizes the result.

Years later, A. J. Yu, G. Ciardo, and G. Lüttgen improved [53] the algorithm mentioned above. Their goal was to be able to perform bounded reachability checking, i. e., to be able to explore a bounded part of the state space. They examined multiple decision diagram formalisms to store the state space, namely Algebraic Decision Diagrams (ADD) and Edge-valued Decision Diagrams (EDD). In addition, they proposed a method that uses ordinary MDDs. According to their measurements, in most cases the EDD-based algorithm provided the best results (both in time and memory consumption).

The bounded EDD-based algorithm in [53] is similar to the former unbounded EDD-based algorithm in [14]. The main difference between them is that the newer algorithm explores only a bounded part of the state space, therefore the states out of bound have to be eliminated from the state space. To achieve this, after every step in the state space (by **SatFire** or **SatRecFire** methods), a *truncation* step is added that is introduced in detail in the next part.

2.6.1 Truncating

The goal of the truncation operation is to prevent (or cut) states outside the given bound to be added to the bounded state space. It is enforced by the **Truncate** method that gets an EDD edge $\langle v, p \rangle$ as a parameter. If the represented subgraph is inside the bound, it returns $\langle v, p \rangle$. If it is out of bound, it returns $\langle \infty, \perp \rangle$ which is the symbolic representation of the empty set.³ As the result of **SatFire** or **SatRecFire** will be substituted with their truncated version, the out-of-bound results will not be added to the state space.

³On level $k \geq 1$ it returns $\langle \infty, z \rangle$, where z is a “zero node” with all edges pointing towards \perp . It is semantically equivalent to the edge $\langle \infty, \perp \rangle$ but with regards to the quasi-reduced form.

Yu et al. introduced two different strategies for truncating: approximate and exact strategy.

Approximative truncation strategy. The *approximative strategy* (`TruncateApprox`) does not enforce an exact bound. Using this strategy we will know, that for a model partitioned into K submodels, all states inside the given bound B are presented in the state space $\mathcal{S}_{b,B}$ and no states are presented outside the bound $K \cdot B$. The advantage of this method that it can be easily done, there is no need to check the subgraph of the given p node, therefore this method is not recursive. Algorithm C.15 shows the implementation of this strategy (the parameter B is marked as *bound* in the pseudocode).

Exact truncation strategy. Using the *exact strategy* (`TruncateExact`), the regular bounded state space exploration can be done, i.e., a global state \mathbf{s} is presented in the state space iff its distance is less or equals to the given bound B ($\delta(\mathbf{s}) \leq B$). This truncate method is not local, it needs recursive computations (see Algorithm C.16).

The results of Yu et al. showed [53] that in most cases, the exact truncate strategy is significantly (by 1–3 magnitudes) slower than the approximate strategy.

Cached exact truncation strategy. However, we (A. Vörös, T. Bartha, and myself) found out that the exact truncation strategy can be competitive with caches introduced [50, 51]. We have observed that the descendants of the result node of the truncation operation will not be modified after the truncation. Therefore the truncate method will return the same truncated edge for the same given edge, thus the computation of `TruncateExact` method can be cached. Our measurements [51] showed that cached exact truncate method is competitive with the approximate method. Also, it can be even faster than the approximate strategy, depending on the characteristics of the analysed model.

It is important to note, that the algorithms presented in [14, 53] assumed that the local state spaces are known a priori. In [51] we showed that the on-the-fly local state space exploration used in unbounded saturation can be used in this case as well.

2.6.2 Iterative bounded state space exploration

It is a natural need to be able to explore the state space with incremental bounds. This is needed for bounded model checking, as during the bounded model checking the state space is explored with bigger and bigger bounds, until the given formula will be evaluated successfully. However, the traditional saturation-based methods are not really suitable to continue the computation with an increased bound. For example, a node can be saturated with a bound B , but using bound $B + 1$, there might be new enabled transitions and reachable states, so it is unknown that the same node is saturated or not. Therefore, it is not trivial, how saturation can be efficiently used in an iterative manner.

Since this is the main topic of my thesis, I will discuss the different algorithms in detail in Chapter 3.

2.7 Bounded CTL model checking

The analysis of bounded CTL model checking is our earlier contribution [51].

Generally, the saturation-based CTL formula evaluation algorithms cannot profit from the additional information encoded in an EDD-based state space.⁴ Therefore, the same algorithms can be used as in unbounded case (introduced in Section 2.5).

To use the same algorithms with the same implementations after bounded state space generation, the state space EDD must be converted to MDD by dropping the encoded distance information. Formally, if the EDD encodes function f_E , the corresponding MDD has to encode f_M [21]:

$$f_M(x_n, \dots, x_1) = \begin{cases} 0, & \text{if } f_E(x_n, \dots, x_1) = \infty \\ 1, & \text{if } f_E(x_n, \dots, x_1) < \infty \end{cases}$$

It has to be noticed that bounded model checking is a *semi-decision* procedure, thus in several cases it can produce an *unknown* result about the examined CTL formula [51]. However, the unbounded CTL formula evaluation algorithms always provide *true* or *false* answer. Earlier we showed that without introducing new saturation algorithms, their result can be extended to use three-valued logic which is able to represent the *unknown* value. It is not connected strongly to the subject of this thesis, but the interested reader can read about this extension in [21, 51].

⁴While the introduced CTL formula evaluation algorithms cannot use this additional information, in some cases it can be useful. For example, shortest path can be generated from initial state to a specific state, which can provide a witness for reachability problems. [14]

Chapter 3

Incremental bounded state space exploration algorithms

This chapter introduces and compares the different methods that can be used to perform incremental bounded states space exploration.¹

In Section 3.2, I describe a simple approach that enables us to execute incremental bounded state space generation and model checking. This method was developed by András Vörös, Tamás Bartha and myself based on former saturation algorithms, introduced in [50, 51]. As I know, there is no other approach published before 2013.

The rest of this chapter introduces my contributions, the *incremental* iterative bounded state space exploration algorithms. The Section 3.3 introduces an extended version of the previous approach, named *continuing saturation*. Section 3.4 is dedicated to the main contribution of this thesis, the so-called *compacting saturation*.

All the bounded state space exploration algorithms described here use the same data structures, namely MDDs for storing next-state functions and EDDs to store state space extended with distance information. All of the following algorithms are *iterative* algorithms. It means that their goal is to explore bigger and bigger part of the state space with an incrementing bound B . The base of all iterative algorithm is the *fixed bound* algorithm presented in Section 2.6 which can explore a bounded part of the state space.

The difference between the algorithms is the amount of data that can be kept between the iterations. Ideally all the results of iteration n should be kept and reused in iteration $n + 1$. However, due to the optimized algorithms some data will be obsolete at the end of the iteration and should be removed. The next section overviews the different data structures that can be reused or dropped at the end of each iteration. After that the different algorithms are introduced and discussed.

¹The restarting algorithm in Section 3.2 is a joint work of A. Vörös, T. Bartha and myself. This was a former work and it is not part of the new results in this thesis. The algorithms presented later in this chapter are my own work and these are the contributions of this thesis. All algorithms presented here is my own work with the help of my supervisors. In the following in this chapter, “we” indicates my thesis supervisors and myself.

3.1 What data can be kept?

As it was discussed, the main problem of incremental saturation-based algorithms is that it is hard to determine, which data remains valid after an iteration.

Saturation data. The basic bounded saturation method extracts and uses the following data:

- local state spaces (\mathcal{S}_k for every level k),
- next-state functions for each event (\mathcal{N}_e for every event e),
- reachable state space (encoded by EDD, denoted by \mathcal{S}),
- caches
 - cache of **SatRecFire** method,
 - cache of **Saturate** method,
 - cache of **Truncate** method,
 - cache of MDD and EDD operations (union, intersect)

In the following, for every data above it has to be determined whether they can be kept and reused in a next iteration or not.

3.2 Restarting algorithm

In 2011, we performed an experiment: can the bounded saturation-based state space generation and the normal saturation-based CTL formula evaluation algorithms be used together? The result of this experience is the restarting algorithm which iteratively uses the bounded state space generation and the normal formula evaluation. To be able to do this, we had to do slight improvements and additions. Our first experiences are described in [50].

The restarting algorithm is the “simplest” way to perform incremental state space exploration based on the fixed bound algorithm. Basically, if the objective is to explore the state space with an increment i , it will first explore the $[0; i]$ part (i.e., states \mathbf{s} such that $\delta(\mathbf{s}) \in [0; i]$), then the $[0; 2 \cdot i]$ part, etc. During the k th iteration, the $[0; k \cdot i]$ part will be explored.



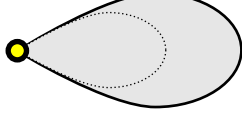
	State space (\mathcal{S})	Part encoded by \mathcal{S}
Iteration 1		$[0; b]$
Iteration 2		$[0; 2 \cdot b]$
Iteration 3		$[0; 3 \cdot b]$

Figure 3.1: *Illustration of restarting algorithm*

Data kept between iterations. A basic implementation of this algorithm drops every saturation data after each iteration. However, it is not necessary to drop all of them, some data will not be obsolete at the end of an iteration, thus they can be kept. In the following, every data element will be discussed.

- The local state spaces can be reused. All local state $i \in \mathcal{S}_k$ that are reachable within bound B are reachable using a bigger bound $B' > B$. Furthermore, all confirmed local states will be confirmed with an increased bound. The unconfirmed states of bound B can be confirmed with bound $B' > B$, and further states can be found and confirmed.
- The partial next-state functions can be reused, but they might need to be extended as new state transitions can be discovered.

Note that the next-state functions of saturation algorithms contains some state transitions that can never be fired. This is due to the on-the-fly local state space exploration. Because there can be state transitions with confirmed “from” state and unconfirmed “to” state, the building of the next-state function has to be continued.

- The state space of the last iteration is dropped. As it is not known, which nodes have to be saturated again, the simplest solution is to drop all of them and rebuild the whole state space from the initial state.
- Because all of the state space nodes are dropped, all caches have to be cleared too. (It is not necessary, but there will not be any cache hits, because the node identifiers are unique. Therefore keeping the contents of caches is just waste of memory.)

An illustration of the restarting algorithm can be seen in Figure 3.1. The parts filled with yellow (with thick border) means the initial state sets. The grey parts are the newly explored states. As it can be seen, the state space exploration are restarted from the initial state(s) in every iteration, and every state is explored newly in each iteration.

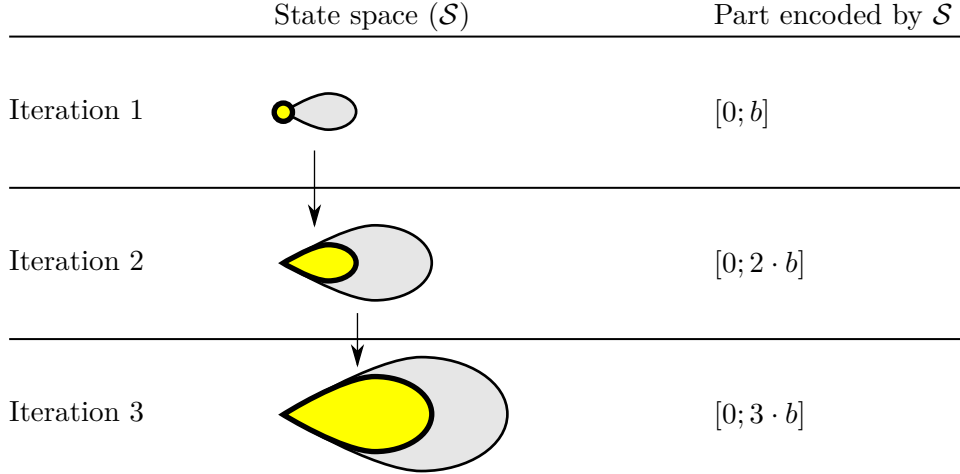


Figure 3.2: *Illustration of continuing algorithm*

3.3 Continuing algorithm

At first try, the restarting algorithm showed that the former saturation-based algorithms can be used together. However, this algorithm was not incremental. It was my motivation to make improvements on it and the first step was the creation of the continuing algorithm.

The aim of the incremental algorithms is to reuse the data from the previous iterations. It seems a big waste for first to restart the exploration of the state space from the initial state. Even if the restarting method can provide a good performance (see for example the search algorithms in the field of artificial intelligence, where the search algorithms restarting every iteration from the initial state are common), it is worth to check whether this restarting method can be improved or not.

The *continuing algorithm* is an advanced version of the restarting algorithm. Its goal is to keep the explored state space between iterations. It is possible, because for all bounds $B < B'$, $\mathcal{S}_{b,B} \subseteq \mathcal{S}_{b,B'}$ holds. (Not only the states of $\mathcal{S}_{b,B}$ are presented in $\mathcal{S}_{b,B'}$, but obviously their stored shortest distances are the same too.)

The continuing algorithm represents the straightforward idea, when the initial state set of the iteration $n + 1$ is the set of explored states in iteration n . An illustration of this algorithm can be seen in Figure 3.2. The parts filled with yellow (with thick border) means the initial state sets. The grey parts are the newly explored states.

However, it is not known, from which states are there any possible new states, in other words which states have to be saturated again. But it has to be noticed that the “re-saturation” of a previously saturated node is not a problem, it is just an unnecessary operation (which does not modify the node). Therefore the continuing algorithm assumes that all nodes from the last iteration are unsaturated and all of them will be saturated in a bottom-up manner.

Generally, the saturation-based state space exploration algorithms working on ordinary Petri nets assumes that there is only one initial state. While this is a valid assumption,

the continuing algorithm needs to run from previous, partial state spaces. Therefore the handling of multiple initial states had to be solved.

3.3.1 Handling set of initial states

To be able to handle a set of initial states, the previously discussed **Saturate** method has to be modified. In the published saturation-based state space exploration algorithms, the creation of the initial state and the saturation of the nodes were interleaved (see Algorithm 3.1). In this way, the nodes can be easily saturated in the needed “bottom-up” manner.

Algorithm 3.1: Interleaved initial state creation and saturation

```

1  $last \leftarrow \perp$ ; // terminal node
2 for  $k = 1$  to  $K$  do
3   Confirm( $k, 0$ ); // confirms initial local state
4    $MDDNode\ n \leftarrow NewNode(k)$ ; // creates new node on the same level
5    $n[0] \leftarrow \langle 0, last \rangle$ ; // initialization of 0th child
6   Saturate( $n$ ); // saturation the new node  $n$ 
7    $n \leftarrow CheckIn(k, n)$ ;
8    $last \leftarrow n$ ;
9 return  $l$ ;

```

Instead of this method, the initial state building (if needed, before the first iteration) and its saturation have to be separated. The building of the decision diagram representation of the initial state (which is an EDD storing only the global state $s_0 = (0, 0, \dots, 0)$ with $\delta(s_0) = 0$) is the same as the former algorithm, but without the **Saturate** call (see Algorithm 3.2).

Algorithm 3.2: Building of initial state

```

input :  $r$ : EDDNode
1 //  $r$ : root node of the initial state
2 EDDNode  $last \leftarrow \perp$ ;
3 for  $k = 1$  to  $K$  do
4   Confirm( $k, 0$ ); // confirms initial local state
5    $EDDNode\ n \leftarrow NewNode(k)$ ; // creates new node on the same level
6    $n[0] \leftarrow \langle 0, last \rangle$ ; // initialization of 0th child
7    $n \leftarrow CheckIn(k, n)$ ;
8    $last \leftarrow n$ ;
9 return  $last$ ;

```

After, if the initial state is created or it is given from the last iteration, it has to be saturated in a bottom-up manner. Before saturating a node, the prerequisite is that all of its children nodes should be saturated. Therefore the saturation has to be started at level 1. But typically the root node is given, thus the decision diagram has to be explored in a depth-first manner (this is the “top-down” saturation).

However, a node can be reached on multiple paths from the root node. For performance reason, every node (and their subgraph) has to be saturated only once. It can be ensured

using a cache storing the already saturated nodes. As saturation (the **Saturate** method) updates the nodes in-place, there is no need to store a reference to the result of the saturation, therefore the cache needs to store only the identifiers of the already saturated nodes and their incoming labels (as it is used for truncating states and the saturation of the same node with different incoming labels can produce different subgraphs).

The pseudocode of the modified method can be seen on Algorithm C.11 in the Appendix.

Note: the idea of the above introduced “top-down” saturation instead of “bottom-up” saturation was presented in [15] as it is needed for CTL formula checking too. However, the “top-down” saturation was not previously written for state space exploration methods, nor for state spaces extended with distance information.

Data kept between iterations. The advantage of this algorithm is that it can keep more data between the iterations than the restarting algorithm can.

- The local state spaces can be reused (as in the case of the restarting algorithm).
- The partial next-state functions can be reused (as in the case of the restarting algorithm).
- The state space of the last iteration is not dropped. Instead, all nodes are saturated again.
- As the state space nodes are not dropped, it is not needed to clear all caches. The operation caches (cache of union and intersect decision diagram operations) can be kept, as their items have not expired. But the bound changed, thus the fire cache and the truncation cache have to be cleared. (For performance reasons, the truncation cache contains only the node-label pair to truncate and its result, but not the used bound value.)

3.4 Compacting saturation

This section is dedicated to my new, iterative saturation-based algorithm, the so-called compacting saturation.

Motivation. The two previously overviewed algorithms have different strengths and according to the measurements: for a part of the models, the restarting strategy is better, for other models, the continuing algorithm is the faster. But both algorithms have a common weakness: these algorithms store the $[0; B]$ part of the state space in one single EDD (where B is the current bound). If the model checking can be finished in a few iterations, it is not a crucial problem. However, if many iterations are needed to evaluate the requirement, the decision diagram describing the state space can be huge. It is an even

bigger problem for EDDs, as the EDD representation of a state space is always larger or equally large as the same state space in MDD². Therefore my goal was to develop a new incremental algorithm that can reduce the size of the EDDs during the analysis.

3.4.1 Main idea of compacting saturation

The observation written above leads to the main idea of the *compacting saturation*. If we partition the state space and we store the result of each iteration individually, it can result smaller, easier-to-handle decision diagrams instead of building a single huge, complex EDD representing the whole explored state space. This is just a hypothesis, it has to be verified.

So formally the idea is to store the state space explored in each iteration separately. The state space explored in the first iteration will be stored in the EDD $\mathcal{S}_{\text{iter}=1}$, the new states explored in the second iteration will be stored in a different EDD $\mathcal{S}_{\text{iter}=2}$, etc.

It is obvious that the EDD $\mathcal{S}_{\text{iter}=1}$ will store the states with distance in $[0, i]$ (i. e., the $[0, i]$ part of the state space). But what will be stored in the next iteration? There are two options: (i) the $[i + 1, 2 \cdot i]$ part, or (ii) the $[i, 2 \cdot i]$.

The first option seems to be more memory efficient, as every reachable global state will be stored exactly once. But as the former saturation methods gets the input from the same EDD where the output will be placed, it is easier to implement the second option. Furthermore, if the decision diagram implementation supports “EDD forests”, the different EDDs can share subgraphs too, as a result, the overlapped part of the EDDs will not be stored in the memory twice.

For that reason, during the k th iteration, the partial state space $\mathcal{S}_{\text{iter}=k}$ created by compacting saturation will contain the $[(k - 1) \cdot i; k \cdot i]$ part.

3.4.2 Overview of the challenges

As the reader can see, the first iteration of the compacting saturation is exactly the same as the first iteration of the restarting or the continuing algorithm. But the k th iteration ($k \geq 2$) is different. The challenges caused by this difference are introduced in this section.

Initial states set. For every saturation-based algorithm, the initial state has to be defined. The first iterations of all bounded algorithm use the encoded form of the model’s initial state. In the k th iteration ($k \geq 2$), the restarting algorithm uses the initial state again, while the continuing algorithm uses the state space of the previous iteration. In

²It can be proved easily. A (quasi-reduced) MDD can be transformed into a (quasi-reduced) EDD by adding labels 0 to every edge. Therefore the size of MDD cannot be greater than the EDD representing the same set. Contrarily, we can show EDDs, that are bigger than the MDDs representing the same set (without distance information). Thus the size of the EDD is always greater or equals to the size of the MDD representing the same set.

the case of the compacting saturation, to ensure that the k th iteration explores the $[(k - 1) \cdot i; k \cdot i]$ part of the states, it has to be started from the set of states whose distance is exactly $(k - 1) \cdot i$. All of these states are known from the previous iteration. It is obvious that for every state \mathbf{s} and for every $\mathbf{s}' \in \mathcal{N}(\mathbf{s})$, the $\delta(\mathbf{s}') \leq \delta(\mathbf{s}) + 1$ (informally: in one single step the minimal distance from the initial state cannot grow by more than one). As the saturation algorithm fires events exhaustively, after the $[(k - 1) \cdot i; k \cdot i]$ part is explored, there is no need for the $[(k - 1) \cdot i; k \cdot i - 1]$ part, as all new possible states can be reached from the states at exactly $k \cdot i$ distance.

Thus the first task of each (noninitial) iteration is to determine the set $\{\mathbf{s} \in \mathcal{S} | \delta(\mathbf{s}) = (k - 1) \cdot i\}$ which will be the initial state set of the iteration.

Iterations. After that, the state space have to be explored with the bound $k \cdot i$. This can be done with the previously mentioned fixed bound state space exploration algorithm, but it introduces a problem. It might happen that previously explored states will be explored again. It is inefficient and because a greater distance will be assigned to it (as all noninitial states in iteration i will have greater assigned distance than states explored in iterations $1, \dots, i - 1$), it can cause wrong results.

To summarize the description above, there are two main challenges in the development of compacting saturation:

- Challenge 1: producing the subset of a state space encoded by EDD which contains only states with distance $\delta = C$ (for a given C).
- Challenge 2: preventing the “reexploration” of states explored in previous iterations.

The Section 3.4.3 shows a solution for Challenge 1, and Section 3.4.4 introduces a possible solution for Challenge 2.

An illustration of the compacting algorithm can be seen in Figure 3.3. The parts filled with yellow (with thick border) mean the initial sets of states. The grey parts are the newly explored states. The border of the state space in each iteration is represented by blue colour. As it can be seen, the border of the state space will become the initial state set of the next iteration. The rest of the state space will be also used (as negated constraint, see later) to solve Challenge 2. It can be also seen in the figure that in each iteration, only the newly explored states are stored in the state space.

3.4.3 Computing the border of the state space

To implement the idea presented before, it is necessary to be able to produce the set of states at a given distance C , as the initial state set of the iteration $n + 1$ will be the subset of states in iteration n that are at maximal distance (i.e. the states that are on the “border of the state space”).

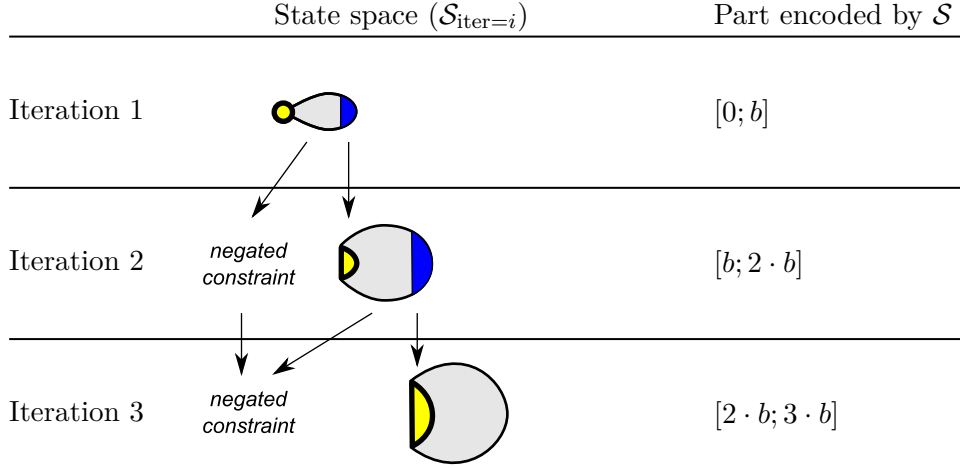


Figure 3.3: *Illustration of compacting algorithm*

The problem discussed here is formally the following: Given a bounded state space \mathcal{S}_b (encoded in EDD, extended with distance information) and a distance C . Produce \mathcal{S}'_b which contains $\mathbf{s} \in \mathcal{S}_b$ iff $\delta(\mathbf{s}) = C$.

As in this case, the given C is maximal distance encoded in \mathcal{S}_b , we can say that the goal is to get the *border* of the state space \mathcal{S}_b .

How can we produce that EDD \mathcal{S}'_b ? Intuitively, it is not possible without the traversal of the decision diagram. A previously described algorithm comes to mind that performs a similar task: the **TruncateExact** method. Basically, it creates a subset of a given EDD that contains only those states that are at distance $\leq B$ from the initial state (for a given B). (Note: the **TruncateExact** method does not modify the given EDD, instead it builds a new diagram.)

This method had to be modified in order to omit all encoded states that are not at distance C . Therefore, if the terminal node is reached by the truncating algorithm and the sum of labels on the path from the EDD root node to the terminal node does not equal to the given distance C , the state must not be presented in the new EDD. Otherwise, the path will be included in the new EDD. The Algorithm 3.3 shows the modified **TruncateExactEq** method. The new lines added to the **TruncateExact** (Algorithm C.16) are marked with asterisks (“*”).

In this way, the **TruncateExact**($\langle v, p \rangle, C$) builds a new EDD that contains all global states \mathbf{s} presented in the subgraph of $\langle v, p \rangle$ such that $\delta(\mathbf{s}) = C$.

3.4.4 Avoiding the redundant computations

As it was mentioned in Section 3.4.1, it is desired to prevent the “reexploration” of states. It means that it is forbidden to discover a global state \mathbf{s} during the iteration i if it was already discovered in a previous iteration $j < i$. It is needed both for efficiency and correctness.

Algorithm 3.3: TruncateExactEq

```
input   :  $\langle v, p \rangle : \text{EDDEdge}, C : \text{int}$ 
output  : EDDEdge

1 if  $v > C$  then
2   // It is impossible to find any states with  $\delta = C$  in this subgraph.
3   return  $\langle \infty, \perp \rangle$ ;
* 4 if  $p.\text{level} = 0 \wedge v \neq C$  then
* 5   // This path encodes a global state and its distance  $\neq C$ .
* 6   return  $\langle \infty, \perp \rangle$ ;
7 if  $p.\text{level} = 0 \wedge v = C$  then
8   // This path encodes a global state but its distance  $= C$ .
9   return  $\langle v, p \rangle$ ;
10 if CacheFind(TRUNCEQ,  $\langle v, p \rangle$ , out  $p'$ ) then
11   return  $\langle v, p' \rangle$ ;
12  $n \leftarrow \text{NewNode}(p.\text{level})$ ;
13 foreach  $i \in S_{p.\text{level}}$  do
14    $r \leftarrow \text{TruncateExactEq}(\langle v + p[i].\text{value}, p[i].\text{node} \rangle)$ ;
15    $n[i] \leftarrow \langle r.\text{value} - v, r.\text{node} \rangle$ ;
16 if  $p$  was checked in then  $n \leftarrow \text{CheckIn}(p.\text{level}, n)$ ;
17 PutIntoCache(TRUNCEQ,  $\langle v, p \rangle, n$ );
18 return  $\langle v, n \rangle$ ;
```

Example 7. To illustrate this problem, consider the Petri net in Figure 3.4. It has an infinite state space which can be seen in Figure 3.5(a). The state \mathbf{s}_i represents the state when $M(p_1) = i$.

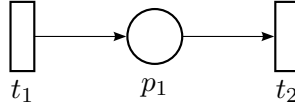


Figure 3.4: Example Petri net to illustrate negated constrained saturation

The execution of the compacting saturation without negated constrained saturation is illustrated by Figure 3.5. During the first iteration, \mathbf{s}_1 and \mathbf{s}_2 are explored. (The states explored during the current iteration are filled with grey. The initial states of each iteration are marked with thick border.) Because \mathbf{s}_2 is on the border of the state space, this state will be the initial state of iteration 2. But because \mathbf{s}_1 is reachable from \mathbf{s}_2 by one transition, it will be reexplored in iteration 2 (with $\delta(\mathbf{s}_1) = 3$ which is wrong). The same is true for \mathbf{s}_0 . Therefore after iteration 2, there will be two states on the border of the state space: \mathbf{s}_0 and \mathbf{s}_4 . Thus the iteration 3 will start from \mathbf{s}_0 and \mathbf{s}_4 too, but exploring the state space from \mathbf{s}_0 is useless and unwanted.

An obvious solution is to execute iteration i as usual, and after eliminate the “wrong” states by subtracting them. It solves the problem of correctness but in a quite inefficient way, as the intersection operation on EDDs is costly. Thus this solution could solve the problem of correctness, but not the problem of efficiency.

This situation resembles the case of EU operator described in Section 2.5. Its first implementation explored more states than needed which were eliminated using an intersection operator. Because it was inefficient, a new approach came into mind: the constrained

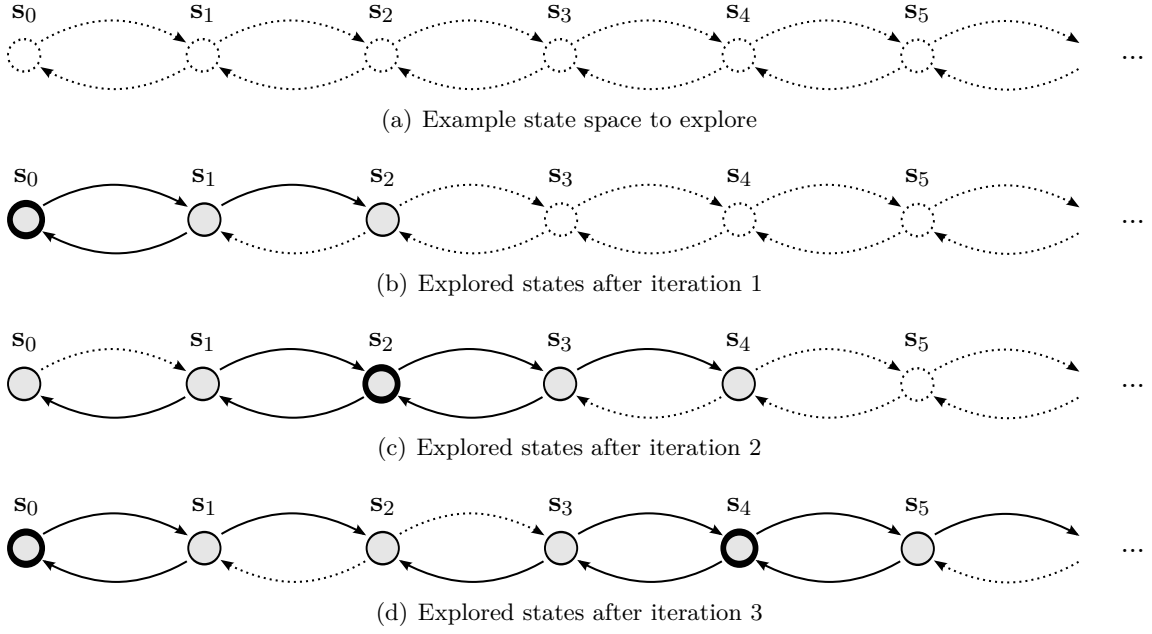


Figure 3.5: *Example for compacting saturation without negated constrained saturation*

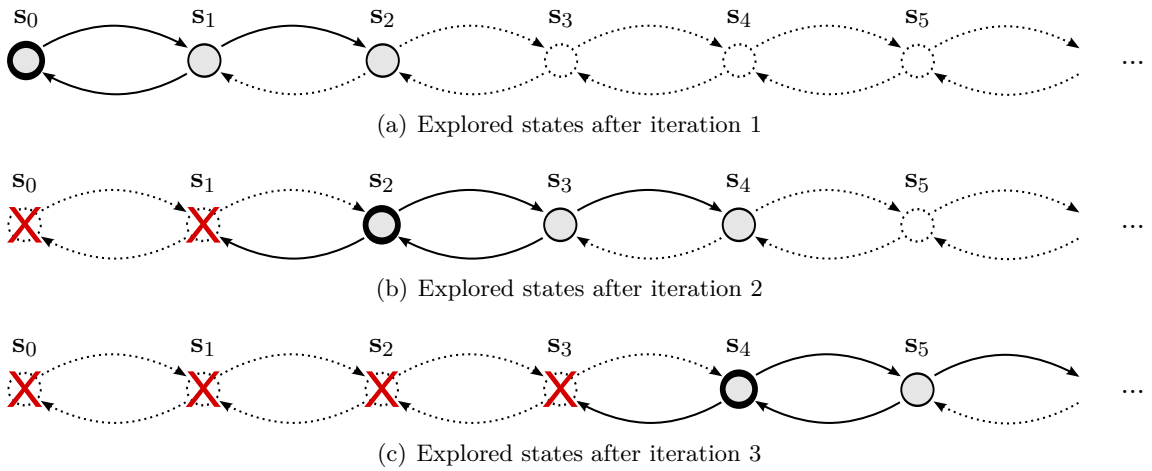


Figure 3.6: *Example for compacting saturation with negated constrained saturation*

saturation. Using this method, it was ensured during the exploration that no unnecessary states were added to the state set.

The constrained saturation ensures that no states are explored out of a given state set. However, to be able to implement compacting saturation, it has to be ensured that no states are explored presented *in* a given state set (i.e., in the set of previously explored states). So the original constrained saturation algorithm cannot be applied directly, but the idea can be reused.

My modified constrained saturation algorithm is the *negated constrained saturation*. It ensures that no states can be inserted to the explored state space that are present in a given set (encoded by a decision diagram). The negated constrained saturation is based on an observation similar to the described in [54]. Let s be an EDD edge encoding a state and let r be the node encoding the corresponding next-state function. Let $negCons$ be the negated constraint corresponds to s . Then the observation formally the following:

$$\mathcal{B}(t) = RelProd(s, r) \setminus \mathcal{B}(negCons) \Leftrightarrow \mathcal{B}(t[i']) = RelProd(s[i], r[i][i']) \setminus \mathcal{B}(negCons[i'])$$

It means that the subtraction operation can be applied during the state space exploration, it is not necessary to perform it after the exploration, using an expensive decision diagram operation. As it is similar to the key observation of constrained saturation, the same methods can be applied here.

Therefore, along with the traversal of the state space EDD, the set of forbidden states (the *negated constraint*) is traversed too. The negated constraint is encoded by an MDD. Before inserting a new global state to the state space EDD, the negated constraint is checked. If the new global state is present in the negated constraint, it will be filtered out from the current EDD. This decision can be made only on the lowest nonterminal level (level 1). If **SatRecFire** would set the current s node's i th edge to \perp with a finite label, but for the negated constraint node c that corresponds to s : $c[i] = \mathbf{1}$, then the new global state is forbidden by the constraint and the edge $s[i]$ will be set to $\langle \infty, \perp \rangle$ (which means the new state is not part of the set).

Because the decision that a global state is permitted or not can only be made at level 1, all **SatRecFire** calls have to recursively continue until it reaches level 1 (or the result is available in the cache). It has to be noted that it exposes overhead compared to the previous iterative algorithms.

The set of forbidden states in iteration i is the set of all states explored in the iterations $1, \dots, i - 1$. Thus the negated constraint is $N = \mathcal{S}_{iter=1} \cup \dots \cup \mathcal{S}_{iter=i-1}$. (Note: because only the forbidden states are needed, their distance information is unnecessary, they can be dropped by converting EDDs to MDDs.)

Example 8. *If negated constrained saturation is used for the same model as in Example 7, the already explored states cannot be included in the state space. The execution of the same*

state space exploration with negated constrained saturation can be observed in Figure 3.6. It provides the wanted solution.

3.4.5 How does the compacting saturation work?

The first state space exploration iteration of the compacting saturation is the same as the first iteration of the restarting or continuing saturation. After this state space exploration, the necessary data for the next iteration is produced: the negated constraint and the new initial state set.

The initial state space of the i th iteration consists of the states on the border of the state space produced during the $(i - 1)$ th iteration. The negated constraint is the MDD representation of all previously explored state spaces.

The pseudocode of compacting saturation can be seen in the Appendix, on Algorithm C.20.

Data kept between iterations. The advantage of this algorithm is that it can keep more data between the iterations than the restarting algorithm can. Also, it can reduce the memory consumption.

- The local state spaces can be reused (as with the restarting and continuing algorithm).
- The partial next-state functions can be reused (as with the restarting and continuing algorithm).
- The state space of the last iteration is dropped, only the border of the state space will be kept and saturated again. The remaining part of the state space will be converted to MDD and stored (which can be more compact and which can be used as constraint).
- The state space EDD nodes are dropped, therefore every cache has to be cleared (as they contain expired/wrong or valueless information).

3.4.6 Possible improvement of the compacting saturation

The compacting saturation described above is ready to be implemented. However, there is an easy-to-see alternative version that should be also analysed.

Above I stated that **SatRecFire** calls have to be called recursively until it reaches level 1, because the negated constraint cannot be evaluated above. While it is true, it can be optimized, since the firing of event e does not modify the state space under level $Bot(e)$. Therefore the only purpose of **SatRecFire** calls on event e below $Bot(e)$ is to ensure that no forbidden states are reached by the upper level operations.

$v \setminus w$	$w = \mathbf{0}$	$w = \mathbf{1}$
$v = \mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$
$v = \mathbf{1}$	$\mathbf{1}$	$\mathbf{0}$

Table 3.1: Truth table of terminal MDD node subtraction

It can be done in an easier way, by simply subtracting the set of forbidden states from the state space under $Bot(e)$. If **SatRecFire** is called with event e on node n with corresponding negate constraint node c and $level(n) < Bot(e)$, then the result will be $n \setminus c$ intuitively.

This observation is formally the following: $RelProd(s[i], r_e[i][i']) = s[i]$, if $level(s) < Bot(e)$, because this subgraph of r_e encodes the identity next-state (sub)function. For that, the subtraction operation have to be introduced.

The extended pseudocode is depicted in Algorithm C.14. (The extensions to the basic compacting saturation is marked with different colour.) The subtraction operation itself is discussed in the next section.

Subtract decision diagram operation

In the following, I formally define the subtraction operator for MDD nodes. As on the terminal level, $\mathbf{0}$ represents \emptyset and $\mathbf{1}$ represents some elements, defining the subtraction operation for terminal MDD nodes is intuitive, as it can be seen in Table 3.1. The nodes on the upper levels can be subtracted by recursively applying the subtraction operator.

Formally, the *subtraction* of MDD nodes v and w (taking place on the same level: $level(v) = level(w)$) is the following:

$$v \setminus w = \begin{cases} v & \text{if } level(v) = level(w) = 0 \text{ and } w = \mathbf{0} \\ \mathbf{0} & \text{if } level(v) = level(w) = 0 \text{ and } w \neq \mathbf{0} \\ z & \text{else, where } z[i] = v[i] \setminus w[i] \text{ for all } i \end{cases}$$

The subtraction operator can be interpreted between EDD edges and MDD nodes too. In this case, the distance information stored in EDD is not considered. The result of this operation is an EDD subgraph.

Formally, the *subtraction* of MDD node w from EDD edge $v = \langle a, p \rangle$ (taking place on the same level: $level(p) = level(w)$) is the following:

$$\langle a, p \rangle \setminus w = \begin{cases} \langle a, p \rangle & \text{if } level(p) = level(w) = 0 \text{ and } w = \mathbf{0} \\ \langle \infty, \perp \rangle & \text{if } level(p) = level(w) = 0 \text{ and } w \neq \mathbf{0} \\ z & \text{else, where } z[i] = \langle a, p \rangle[i] \setminus w[i] \text{ for all } i \end{cases}$$

Using these definitions, the subtraction operations can be implemented easily.

3.5 General problems of saturation-based iterative model checking

There are some further problems that concerns all presented iterative variants of saturation-based model checking. This section briefly overviews these problems and the given solutions.

Termination criterion. As I stated before, the iterative model checking explores bigger and bigger parts of the state space. But when can this algorithm stop? There general answer is simple: when the given requirement can be evaluated. It raises another question: when can a requirement be evaluated? There are two main cases: (1) if the full state space is explored (in this case, the result given by the model checker will be certainly valid), or (2) if the model checker can give a certain answer based on a bounded state space exploration.

For example, if the model checker evaluates $\text{EF } p$ to true, it is a certain answer based on a bounded state space exploration. If the $\text{EF } p$ is true for a part of the state space, it will be true for the full state space too. But if the model checker evaluates $\text{EF } p$ to false based on a bounded state space, this answer is not certainly true for the full state space, therefore the examination have to be continued with bigger bound. In this case, it is easy to see, what are the authoritative answers, but it is not true for every CTL operator.

A simple solution —that is used in the current implementations— is to ask the user, what is his/her expected answer. Then the model checker tries to prove the expectation. If the model checking algorithm gives the expected answer, the iterative method will stop.

However, in [21, 51] we proposed exact conditions, when the answer of the iterative saturation-based model checker is authoritative. The drawback of this method is that in certain cases the “authoritativeness check” needs the computation of other CTL expressions which increases the required evaluation time.

Detection of the full state space exploration. Another needed feature is the detection whether the iterative state space exploration reached the full state space or not. When the full state space is already explored, the iterative algorithms should be stopped.

This detection depends on the applied truncating method. If the exact truncating method is used, it is easy to detect the exploration of the full state space. If no new states have been found in the last iteration, there is no need to continue the exploration as there are no unexplored states. But if the approximate method is used, it is hard to detect the end of the exploration, because the fact that there were no new states using bound $B + i$ compared to the result with bound B does not imply that there will be no new states using bound $B + 2i$. Therefore if the detection of full state space exploration is needed, the exact truncating method has to be used.

Even for exact truncation method, there are three different possibilities for detecting the end of the iterative state space exploration:

1. Detect when *the count of states in the state space did not grow* during an iteration. While this is straightforward, it needs the calculation of states encoded by the state space decision diagram which can be expensive.
2. Detect when the *root node of the state space is not modified*. This method is easy to check, but it is only applicable when the state space contains all previous states (and if it is not restarted).
3. Detect when there are *no states on the “border” of the state space* (e.g. there are no states in the state space of bound B (when $\delta = B$), formally: $\{\mathbf{s} \in \mathcal{S}_{b,B} | \delta(\mathbf{s}) = B\} = \emptyset$). Using this method, the end of the iterative algorithm can be detected sooner as it does not need another iteration with no additional states which would prove that the state space is not modified, as in the previous method.

The presented iterative strategy uses different solutions. The third solution is the best, but the most expensive of the three possibilities. However the border of the state space is calculated for compacting saturation, therefore this solution can be used in this iterative variant.

The restarting algorithm builds new decision diagrams for representing state spaces, therefore the second solution cannot be used. Thus, the first solution is implemented for this algorithm. The continuing algorithm iteratively extends the same decision diagram, so the second solution can be applied.

Chapter 4

Realization of the compacting saturation

All the algorithms presented in Chapter 2 and Chapter 3 are implemented in the PetriDotNet framework. This framework is briefly introduced in Section 4.1. The rest of this chapter describes the main details of the implementation (Section 4.2).

4.1 PetriDotNet framework

The *PetriDotNet* framework [55] is a software for editing, simulating and analysing Petri nets. It is developed at the Fault Tolerant Systems Research Group at the Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary. The developers are curious, talented students, working on this project mostly in their free time¹.

PetriDotNet supports multiple variant of Petri nets: ordinary Petri nets; ordinary Petri nets extended with priorities, inhibitor edges, capacity limits; hierarchical Petri nets, well-formed coloured Petri nets, etc.

Its goal is to provide an easy-to-use interface for the users, and also for the developers. This framework can be easily extended with plugins. My current work is also developed as a PetriDotNet plugin.

As an illustration, the main window of the PetriDotNet framework can be see in Figure 4.1.

¹Here I would like to thank everybody who has participated in the PetriDotNet project during the last 5 years, especially the following main contributors: Bertalan Szilvási, Attila Jámbor, Vince Molnár, Attila Klenik. They helped me a lot with their implementations, suggestions, and bug reports to be able to develop the algorithms presented here. My supervisors, Tamás Bartha and András Vörös also helped the development of the framework a lot.

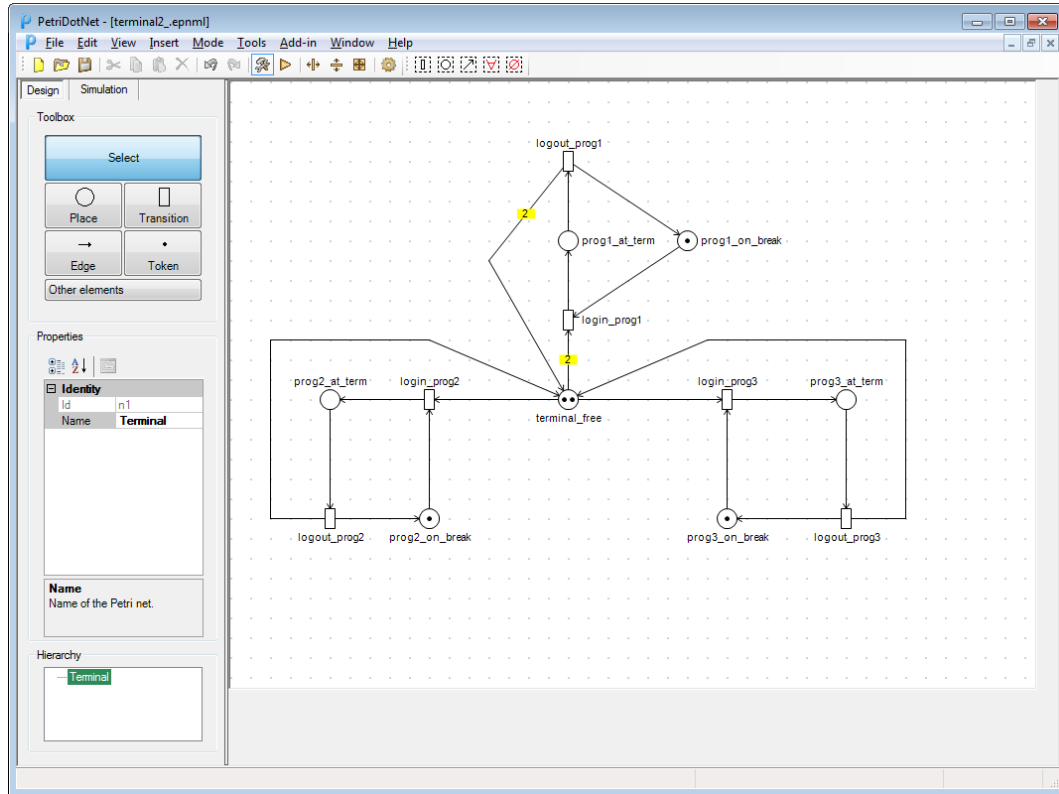


Figure 4.1: *The main window of the PetriDotNet framework*

4.2 Details of the implementation

While my main contribution is the algorithmic and analysis part of this thesis, to be able to measure the performance of the algorithms, I implemented them. In this section, I overview some details of the implementation.

The implementation of the described algorithms itself are simple after their pseudocodes are given. However, to be able to use them, I did a lot of supporting work. The PetriDotNet framework provides a graphical user interface, the handling of the models (loading and saving the models), and an object model of the Petri nets. Thus the analysis plugins do not have to parse or write directly model files. This part of the framework was already developed by Bertalan Szilvási and myself when I started the current work, thus its implementation questions are out of the scope of this thesis. The reader can find details about its initial conception in [48].

The presented analysis modules highly rely on decision diagrams. The used decision diagram library is introduced in Section 4.2.1. Afterwards I present some deeper problems affecting the performance of the implementation, which is crucial in the case of model checking algorithms.

4.2.1 Implementation of the decision diagrams

This subsection overviews the implementation of the decision diagrams. In 2010, when I started to work with saturation-based algorithms, there were no publicly available .NET-based decision diagram implementations. Therefore I developed my own implementation, based on the experiences written in [49]. As it was an earlier work, the introduction here is high-level, the deep implementation details are omitted.²

The two main classes for each diagram type are the **Node** and **Forest** classes (see Figure 4.2).

- A **Node** class (**MDDNode**, **EDDNode**) describes a decision diagram node, which contains its identifier, level number, and its children nodes (or the outgoing edges in the case of **EDDNode**). The node classes contains static methods for the node operations and the **CheckIn** method as it was introduced in [40].
- A **Forest** class (**MDDForest**, **EDDForest**) represents a set of decision diagrams³. It is true that to put each decision diagram into a separate object and handle them as individual entities would respect more the encapsulation principle of the object oriented programming. However, if multiple decision diagrams are stored in the same container, their common, redundant subgraphs are stored only once. But there is a common property of the decision diagrams stored in the same **Forest** object: they have equal number of levels. Without this constraint, the decision diagrams would have different semantics. The forest also stores the terminal nodes, which are common for every decision diagram stored in the forest.

Note: the classes that can be seen on the diagrams are implemented as C# classes. The C# properties are represented as public fields to be easily understandable. In reality, these fields are private and they have public getter and setter methods.

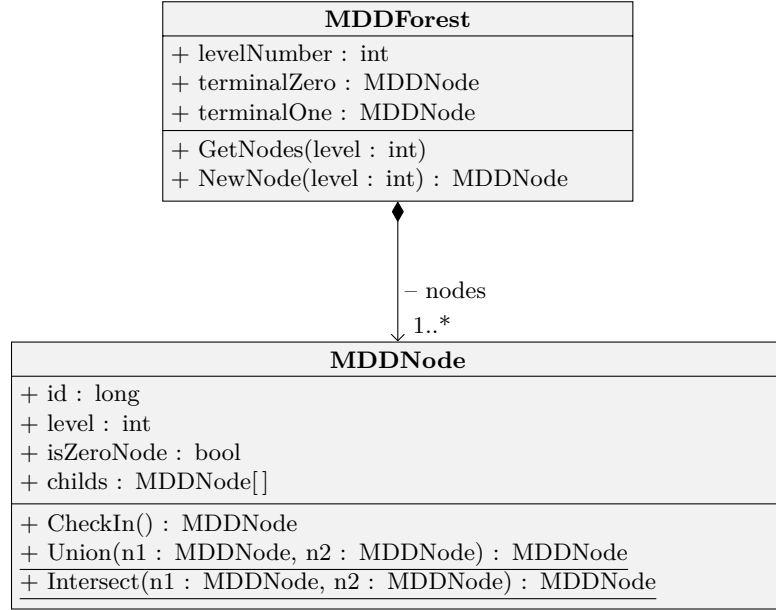
4.2.2 Creating and destroying node objects

During our previous work we discovered that the saturation-based algorithms are highly affected by the performance of the decision diagram implementation. One of the drawbacks is that the saturation algorithms creates and destroys large amount of nodes. On the level of the .NET framework, it means lots of constructor and destructor calls, and also bigger work for the garbage collector.

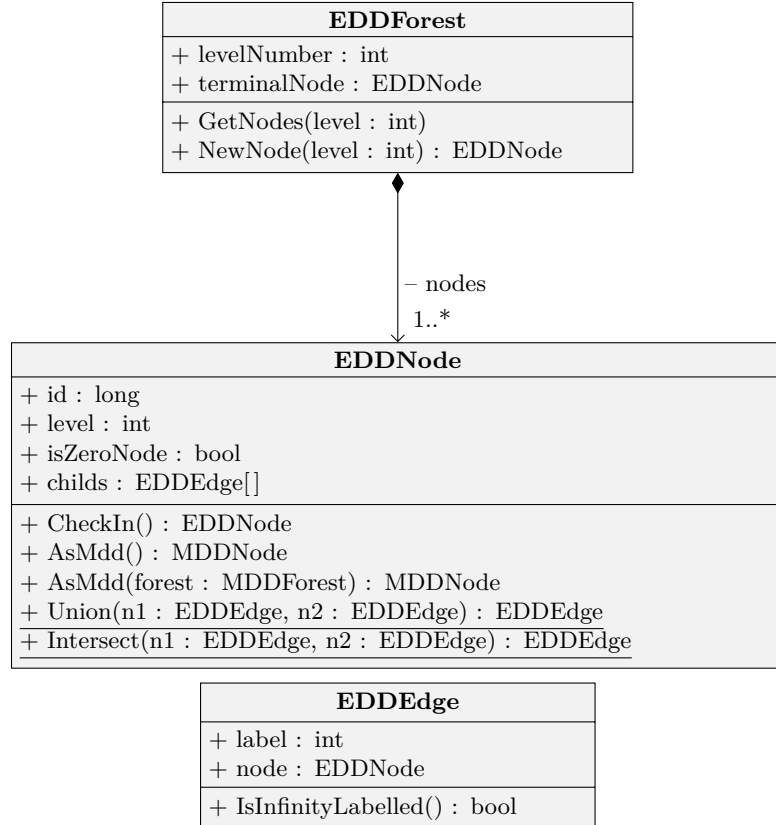
However, instead of destroying an object with its destructor, it can be put into a container (“stack” or “pool”) of nonused objects. When a new node is needed, a previously deleted

²For example, the real implementation uses abstract classes, inheritance, generic classes, multiple abstraction layers, reference counting, automatic node deletion, etc.

³While forest is an intuitive name for this class, it has to be noticed that the decision diagrams are not trees, but directed acyclic graphs (DAGs).



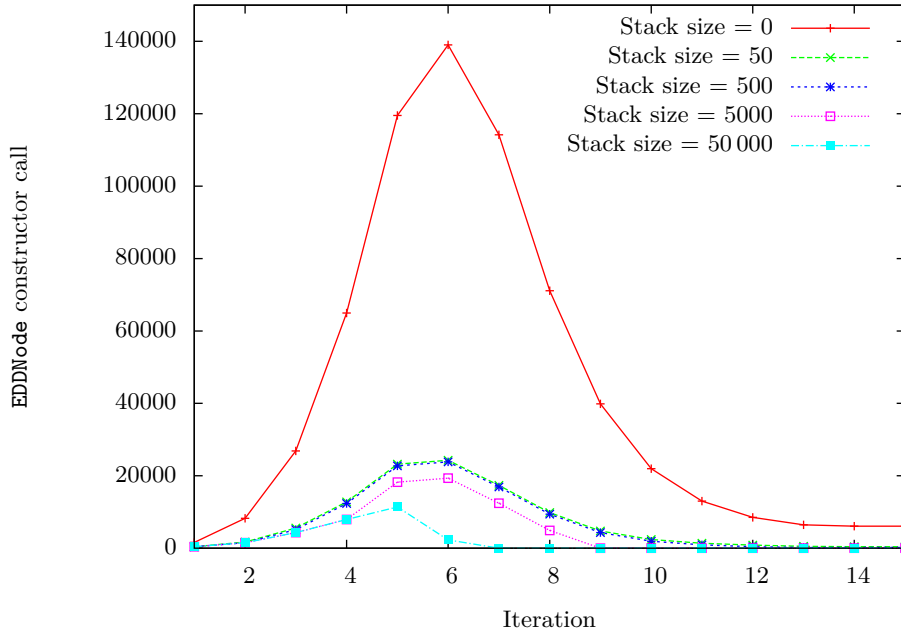
(a) Basic MDD structure



(b) Basic EDD structure

Figure 4.2: *Class diagrams of the decision diagram implementation*

Figure 4.3: *Number of created EDDNode objects in each iteration using different stack sizes*



node can be reinitialized, so no constructor call is needed. The implementation of this idea is quite simple, as it can be seen on Listing 4.1. Note: this solution is similar to the pooling proposed in [32].

To illustrate this solution, I executed a restarting bounded state space exploration (on model FMS-5 with increment = 5) with different stack (pool) sizes. The results can be seen in Figure 4.3. It is easy to see, that even a small-sized stack can reduce significantly the EDDNode constructor calls. Its reason is that there are lots of objects with short lifetime (for example large amount of node created by the union operator is dropped immediately after CheckIn call).

While it is true that bigger object cache means lower number of constructor calls, the same is not always true for the run time. I measured large models with stack size of 500 and 5000 and in most cases, the smaller size produced better results. The reason of this characteristic is not clear. It can be caused by older, lower quality implementations in the saturation module of PetriDotNet, or the optimizations in the .NET garbage collector. For further improvements of the implementation, it would need further studies.

Therefore in the measurements the applied object cache size was 500 which provided better results than the 0 or 5000 value.

4.2.3 Effects of micro-optimization

The .NET framework and the C# language provides a rich and easy-to-use base for software development. These features (e.g., delegates, LINQ expressions, reflection) enables

Listing 4.1: *Cache of deleted nodes.*

```

1 public class MDDNode : IDisposable
2 {
3     /// <summary>
4     /// Cache of old objects.
5     /// </summary>
6     private static Stack<MDDNode> oldInsts = new Stack<MDDNode>();
7
8     /// <summary>
9     /// Maximal size of old instance cache.
10    /// </summary>
11    public static int StackMaxSize { get; set; }
12
13    /// <summary>
14    /// Initializes the static fields (with default values).
15    /// </summary>
16    static MDDNode()
17    {
18        StackMaxSize = 500;
19    }
20
21    /// <summary>
22    /// Returns a new instance of the current class.
23    /// </summary>
24    /// <returns>New node instance.</returns>
25    private static MDDNode GetInstance()
26    {
27        if (oldInsts.Count == 0)
28        {
29            return new MDDNode();
30        }
31
32        var ret = oldInsts.Pop();
33        return ret;
34    }
35
36    /// <summary>
37    /// Returns a new instance of the current class initialized with the given parameters.
38    /// </summary>
39    /// <returns>New node instance.</returns>
40    public static MDDNode GetInstance( [...] )
41    {
42        var ret = GetInstance();
43        ret.Initialize( [...] );
44        return ret;
45    }
46
47    /// <summary>
48    /// Puts the unused instance to the cache.
49    /// </summary>
50    public void Dispose()
51    {
52        if (oldInsts.Count <= StackMaxSize)
53        {
54            // The instance properties and associations are deleted here in the real ↵
55            // implementation.
56
57            oldInsts.Push(this);
58        }
59    }

```


us to write high-quality programs in short time.

However, some of these extensions are not suitable for algorithms with high computational needs, like the saturation-based model checking. It is not extraordinary, if a method of a saturation-based algorithm is called for million or billion times, even for small models that can be verified in a couple of minutes. In this section, I introduce some observations gathered during the development about what structures can be used and what constructions are better to be avoided.

The “First()” LINQ expression

If we want to pick the first element of a list of array, it is common to use the `First()` LINQ method, because it is simple and easy to read. However, after the application of the `First()` method, I experienced slowdown in the execution. The reason is that this method creates first an enumerator which is unnecessary in this simple case. Thus, it wastes time and memory.

According to the measurements (see Table 4.1), an `array.First()` call requires about 150 times more run time than a simple `array[0]` statement (without compilation optimizations).

Table 4.1: *Measurements of indexing and First() call*

Solution	Run time per call [ns]
<code>array.First()</code> call	204.64
<code>array[0]</code> call	1.39

It has to be noted that in C# even the array indexing is more complex than for example in the C language, as the indexing operator is a method call and it contains some over-indexing check. But if we have a look at the disassembled code (see Listing 4.2), even it is a method call, no real `call` CPU instructions will be invoked, as the JIT (Just-In-Time) compiler simplifies the code and eliminates the call in the case of the small methods (this is the so-called “method inlining”). In some cases, if the indexing is safe (the index is always between the bounds), the JIT can totally eliminate this index checking and error handling part too.

Listing 4.2: *Disassembled code of an array indexing*

1	0000005e	mov	eax, dword ptr ds:[03E64F3Ch]	; move the start address of the array to eax ←
		register		
2	00000063	cmp	dword ptr [eax+4], 0	; compare the length of array to the desired index
3	00000067	ja	0000006E	; if index < length, jump after the call statement
4	00000069	call	62073FD7	; error handling (throw ←
			System.IndexOutOfRangeException)	
5	0000006e	mov	eax, dword ptr [eax+8]	; move the array[0] value to eax register
6	00000071	mov	dword ptr [ebp-40h], eax	; move the content of eax to variable x

The disassembly of the `First()` call looks simpler (see Listing 4.3), but here the call is not eliminated by the JIT compiler. The `First()` method itself contains 90 CPU instructions, so it is not be discussed here, but this method is significantly more complex.

Therefore no `First()` calls can be used in the saturation codes, even their usage is intuitive and makes the source code more readable.

Listing 4.3: *Disassembled code of a `First()` call*

```

1 0000005e mov     ecx,dword ptr ds:[03103348h] ; move the start address of the array to eax ←
    register
2 00000064 call    603045A0                    ; System.Linq.Enumerable.First() call (contains ~90←
    instructions)
3 00000069 mov     dword ptr [ebp-54h],eax      ; gets the address of the return object
4 0000006c mov     eax,dword ptr [ebp-54h]    ; gets the return object
5 0000006f mov     dword ptr [ebp-40h],eax    ; move the content of eax to variable x

```

Conditional statements

The base saturation algorithm has many variants and also, they can be parametrized. If every variant has a disjoint implementation, the variants do not have effects on each other, but the source code will not be easily maintainable. If there is only one “unified” source code for all variants, it could simplify the maintenance but it could have impact on the performance.

Usually some conditional statements will not have serious impact on the run time, but typically saturation uses short methods for plenty of times, as it was mentioned before, therefore it has to be examined, what is the impact of conditional statements to the performance of the algorithms.

Conditional blocks to parametrize the execution of the program can be implemented in multiple ways, including:

1. *Conditional compilation symbols.* In this way, the unnecessary part of the source code will not be presented in the compiled code, thus the unnecessary part will not make any impact on the performance.

```

1  #if false
2  UnnecessaryOperation(1000);
3  #endif

```

2. *Conditions given by constants.* In this way, the unnecessary part of the source code can be presented in the compiled code, but it cannot be used.

```

1  const bool COND = false;
2  // ...
3  if (COND)
4  {
5      UnnecessaryOperation(1000);
6  }

```

3. *Conditions given by local variables.* This solution can provide some possibility of reconfiguration, but if the local variable has a constant value, the compiler can optimize and eliminate it.

```

1  public void Test()
2  {
3      bool cond = false;
4      if (cond)
5      {
6          UnnecessaryOperation(1000);
7      }
8  }

```

4. *Conditions given by local fields.* The advantage of this solution is that the algorithm can be reconfigured in runtime.

```

1  public class Test
2  {
3      private bool cond = false;
4
5      public void Test()
6      {
7          if (this.cond)
8          {
9              UnnecessaryOperation(1000);
10         }
11     }
12 }

```

5. *Conditions given by public property of an other object.* The advantage of this solution is that the algorithm can be easily reconfigured in runtime, using the object oriented paradigms.

```

1  public class ConditionDescriptor
2  {
3      public Condition {get; set;}
4      public ConditionDescriptor()
5      {
6          this.Condition = false;
7      }
8  }
9

```

```

10 public class Test
11 {
12     private ConditionDescriptor condDesc = new ConditionDescriptor();
13
14     public void Test()
15     {
16         if (condDesc.Condition)
17         {
18             UnnecessaryOperation(1000);
19         }
20     }
21 }

```

The impact of the different conditional statements is not always intuitive, as the final code can be modified by the Just-In-Time (JIT) compiler. The JIT compiler can reduce code size by “method inlining”, it can eliminate unnecessary conditions, etc. [32].

I have measured the upper five possible solutions by calling them for 1,000,000,000 times. The run time results are the following:

Solution	Run time [ms]	Run time difference from Solution 1 [ms]
(1) Compilation symbol	1533	0
(2) Constants	1523	−10
(3) Local variable	1520	−13
(4) Local field	1906	+373 (+24.3%)
(5) Property of different object	2899	+1366 (+89.1%)

As the measurements showed, there is no difference between the first three solutions.⁴ If we examine the disassembled output of the JIT compiler, the conditional statement will be entirely omitted from the program.

Contrarily, if the condition is given by a local field (Solution 4), some processor instructions will remain, the disassembled conditional statement is the following:

```

1  mov     eax,dword ptr [ebp-10h] ; eax := address of this
2  cmp     byte ptr [eax+4],0      ; compare this.cond and "false"
3  je      AFTER_COND             ; if the condition is false, jump through the ↔
4                                     UnnecessaryOperation() call
5  mov     ecx,3E8h               ; parameter (1000) passing for the call
6  call    5FFB0898               ; UnnecessaryOperation() call
7  AFTER_COND:
8  ...

```

⁴Note that the third solution used constants as values of the local variable, therefore the conditional statement was eliminated by the compiler.

As it can be seen from the disassembled code, every conditional statement implemented using Solution 4 will add 3 additional processor instruction. Usually it is a really small overhead that is completely tolerable.

The measurements showed that Solution 5 (which is the most elegant solution above) is significantly more costly than the others.

```

1  mov     eax,dword ptr [ebp-10h] ; eax := address of this
2  mov     eax,dword ptr [eax+4] ; eax := address of condDesc
3  cmp     byte ptr [eax+4],0      ; compare condDesc.Condition and "false"
4  je      AFTER_COND             ; if the condition is false, jump through the ↔
                                   UnnecessaryOperation() call
5  mov     ecx,3E8h               ; parameter (1000) passing for the call
6  call    5FFB0898               ; UnnecessaryOperation() call
7  AFTER_COND:
8  ...

```

As you can see, the call of getter property has been eliminated by the JIT compiler and only the additional indirection raises the number of necessary processor instructions.

Therefore it is not necessary to implement all variants of the algorithms separately, the algorithms can be parametrized with small time penalty. For performance reasons, the earlier saturation-based algorithms were implemented separately or used compilation symbols. The conclusion of these measurements is that it is not necessary, constants and local variables can be applied to configure the execution of the model checking, thus it is possible to reconfigure the algorithms in runtime.

4.2.4 Usage of delegates

In the .NET framework, another possibility to create general algorithms is the usage of delegates. The delegate is a typed function reference, similar to the function pointer in C. The use of delegates would be handful in bounded algorithms as the bounded state space exploration algorithms can use multiple truncation strategies (exact or appropriate, see Section 2.6.1). But what is the impact of the delegates to the performance?

To be able to determine it, I measured the cost of direct calls and the calls using delegates. The measurement codes can be seen on Listing D.1.3. I also measured the cost of parametrized calls: in this measurement, I added a `long` and a `bool` parameter to the called method.

The run time measurements can be seen in Table 4.2. As it can be seen, there is a significant difference between the cost of direct calls and calls through delegates, even if parameters are used. However, the impact of this difference is usually small, as the direct calls and the calls through delegates are fast, the difference between them in real applications is not significant.

If we have a look at the disassembled code, we can see that the direct call is compiled into a single `call` processor instruction without parameters.

Table 4.2: *Measurements of overhead of delegates*

Solution	Run time per call [ns]	
	without parameters	with parameters
direct call	2.52	2.53
delegate call	3.62	3.90

```

1  call    dword ptr ds:[0052385Ch] ; calls TestMethod directly

```

If delegates are used instead of direct calls, some new processor instructions are added to the compiled code, because the address of the method has to be determined, as it can be seen below.

```

1  mov     ecx,dword ptr [ebp-28h] ; ecx := variable del
2  mov     eax,dword ptr [ecx+0Ch] ; eax := del._methodPetr (address of TestMethod)
3  mov     ecx,dword ptr [ecx+4]   ; ? (in this case, it does not modify ecx)
4  call    eax                    ; calls TestMethod

```

If the call is parametrized, some new instructions are added. To put the long parameter to the stack, two push instructions are needed. The Boolean parameter is passed through a register, it needed one more processor instructions, as it can be seen below.

```

1  push    dword ptr [ebp-0Ch] ; push first part of the long parameter
2  push    dword ptr [ebp-10h] ; push second part of the long parameter
3  xor     ecx,ecx             ; sets the bool parameter (to false)
4  call    dword ptr ds:[0028386Ch]

```

If the same parametrized method call is performed by a delegate, the same additional instructions can be observed (two push and one xor instruction). Therefore the cost of using delegates is not higher if parameters are used.

```

1  push    dword ptr [ebp-10h] ; push first part of the long parameter
2  push    dword ptr [ebp-14h] ; push second part of the long parameter
3  mov     ecx,dword ptr [ebp-28h]
4  xor     edx,edx             ; sets the bool parameter (to false)
5  mov     eax,dword ptr [ecx+0Ch]
6  mov     ecx,dword ptr [ecx+4]
7  call    eax

```

As it can be seen, usage of delegates means certain overhead, but the absolute difference between direct calls and calls through delegates is small, their impact on the execution time of the whole model checking is not significant. Therefore I used delegates in the realization of the bounded saturation-based state space exploration algorithms. In this way, it is easily parametrizable which truncating method is applied.

Chapter 5

Evaluation

In this section measurements are introduced and discussed supporting the evaluation of the presented algorithms. After the introduction, the Section 5.1 shows run time measurements. The next chapter (Section 5.2) presents memory consumption measurements. Section 5.3 is dedicated to the comparison of compacting and noncompacting incremental methods. Section 5.4 describes two real case studies where the new model checking techniques can be applied. Finally, the conclusions are summarized in Section 5.5.

To evaluate the new developments, they have to be compared to the former algorithms. As my task was to improve the saturation-based bounded saturation algorithms, I have chosen the following algorithms as baselines:

- **Unlimited saturation algorithm (Unlim)** This is basically the same algorithm as the one presented in [14] or the algorithm in [53] with infinite bound and without truncation. Using this algorithm, the complete state space will always be explored enriched with distance information.
- **Restarting bounded saturation algorithm (Rest)** This algorithm is the basic incremental bounded saturation presented in [50, 51]. It starts every iteration from scratch, so it runs from the initial state space in each iteration. The detailed introduction of this algorithm can be read in Section 3.2. My primary goal was to improve its performance.

I will compare the following new algorithms to them:

- **Continuing bounded saturation algorithm (Cont)** This algorithm is similar to the restarting algorithm, except it continues every iteration from the state space explored in the last iteration. The algorithm is presented in Section 3.3.
- This is the algorithm presented in Section 3.4 which starts every iteration from the states on the border of the explored state space of the last iteration.

- **Compacting bounded saturation algorithm with subtraction (CompSub)**

This algorithm is the compacting bounded saturation algorithm improved by the extension with subtraction described in Section 3.4.6.

Now it is given which algorithms will be measured. But how they can be measured and compared? As my goal was to improve the *performance* of the former algorithms, I will compare their performance. Generally, there are three main dimensions of performance comparison:

- run time comparison,
- memory consumption comparison,
- I/O consumption (usage of disk, network, etc.).

I/O measurements. As the saturation-based analysis algorithms does not use the network or the disk directly, it is unnecessary to compare them in these dimensions.

Run time measurements. The measurement of the run time is relatively easy. The .NET framework provides an accurate timer class (`System.Diagnostics.Stopwatch`). Using that, the run time of the algorithms can be measured. It has a relatively small performance penalty (there is no resource consumption after starting it except a small amount of memory), while it has high-precision on modern computers.¹

Memory measurements. Measuring the memory consumption is not trivial and ambiguous in the field of managed software. Two different approaches exist to measure the memory consumption:

- measuring the *memory allocation*,
- measuring the *peak or average memory consumption*.

Basically, the memory allocation measures the total cumulated size of all created objects. This measure is deterministic, but it does not determine, how much memory is needed by the application.

If we measure the peak memory consumption, we measure the maximum size of active objects. However, .NET uses a garbage collector to free up the space occupied by unused objects. This garbage collector runs when the operating system is low on memory or

¹The precision depends on the used computer but typically it is about 10^6 to 10^9 tick/second which means 1 μ s to 1 ns accuracy. If the hardware does not support a high-resolution performance counter, `Stopwatch` uses the `DateTime` managed class instead of the unmanaged `QueryPerformanceCounter` Win32 API call. In this case, the accuracy is about 10 to 20 ms.

its *Generation 0* (i.e., the area of young objects in the heap) is full [32], so it is quasi-nondeterministic from the view of the application. It highly depends on the current computer and the other running programs. Therefore the peak memory consumption is difficult to measure and the results are not accurate.²

However, measuring the peak memory consumption is not crucial in this case. Consider an ordinary computer with 4 GiB of RAM. If the algorithm needs significantly less memory than the available memory, it will run (relatively) fast, because there will not be too many need for garbage collection. In the case of a verification, usually it is not primary whether the program consumes 100 MiB or 1 GiB, as typically no other operations will be done on the same computer at the same time.

The more memory the application consumes, the more garbage collection will be necessary. If the peak memory need is larger than the size of the available memory, the performance will be significantly dropped due to the garbage collection and the automatic swapping performed by the operating system. So the memory needs will highly affect the run time of the algorithm, thus the run time of the algorithm is a more important metric than the memory consumption.

In Section 5.1 run time measurements are shown and evaluated both for full state space exploration and for CTL expression evaluation (which is the real task for the iterative algorithms). After, some memory measurements are discussed in Section 5.2.

5.1 Run time measurements

This section compares the run times of the described bounded saturation algorithms on different models with different CTL expressions to evaluate. (The models used in measurements are described in Appendix B.)

Measurement method. The run time of the algorithms are measured as the time elapsed from starting to the finishing of the algorithm. This run time does not include the initialization of the global data structures and the GUI input/output. The time is measured using the previously described `Stopwatch` class. The machine used for measuring is defined in Appendix D.3.

The measurements in this section are divided into two parts: the run time measurements of the full state space exploration and the run time measurements of the evaluation of a CTL expression.

²The reader can find more information about the garbage collector in the .NET Framework in [32]. There are lots of possibilities to tune the garbage collection methods which could be useful in the future for the saturation algorithms.

5.1.1 Full state space exploration

The first dimension of the run time analysis is the full state space exploration. In this case, the complete state space have to be explored by the model checker to be able to evaluate the requirement. This is the worst case scenario for the bounded model checking algorithms. It cannot be expected that any of the bounded model checking algorithms will outperform the non-bounded algorithms, as the bounded algorithms have their typical overhead, but they cannot stop earlier, thus it is not possible to benefit from the boundedness. However, it is important to compare the performance of the different bounded algorithms.

Table 5.1 shows the results of the measurements, done on several different models of different size (different parameters). The bold numbers mark the best run time for each model. If the best run time is achieved using the Unlimited algorithm, then the best bounded run time is bold too. There are some observations easy to remark.

- The run time of the Unlimited saturation algorithm (Unlim) algorithm is less than the run time of the other algorithms in most cases. Its reason is that the incremental algorithms have an overhead due to their incrementality. In a normal case, there is a trade-off between the overhead of the incrementality and the advantage of exploring fewer states. However, if the full state space is explored, there is no advantage of exploring fewer states than all states.
- Contrarily to the first observation, in the case of the Hanoi model, the compacting algorithms perform better than the unlimited algorithm. This anomaly is discussed in details in Section 5.3.
- Usually the results of the restarting and the continuing are close to each other. The same is true for the two compacting methods. However, there are significant differences between the results of the restarting/continuing and the compacting methods.
- For some models, like Phil- N , RR- N , Kanban- N , the compacting algorithm with subtraction extension (CompSub) performs better than the simple compacting algorithm.
- It can be seen that the run time depends not only on the model but on the increment parameter too. (For example, the run time of incremental algorithms are better with increment=2 than with increment=1 for the model Queen-10.) This dependency is discussed below in details.

Effect of the parameters. As I stated above, the results of the measurements depends not only on the models, but on the parameters too. Important parameters are the partitioning of the model and the increment parameter of the bounded algorithm. As the challenges in model partitioning is not discussed in this thesis, I focus on the other parameter. (The partitioning was always the same for each model. The used partitioning is given in Appendix B.)

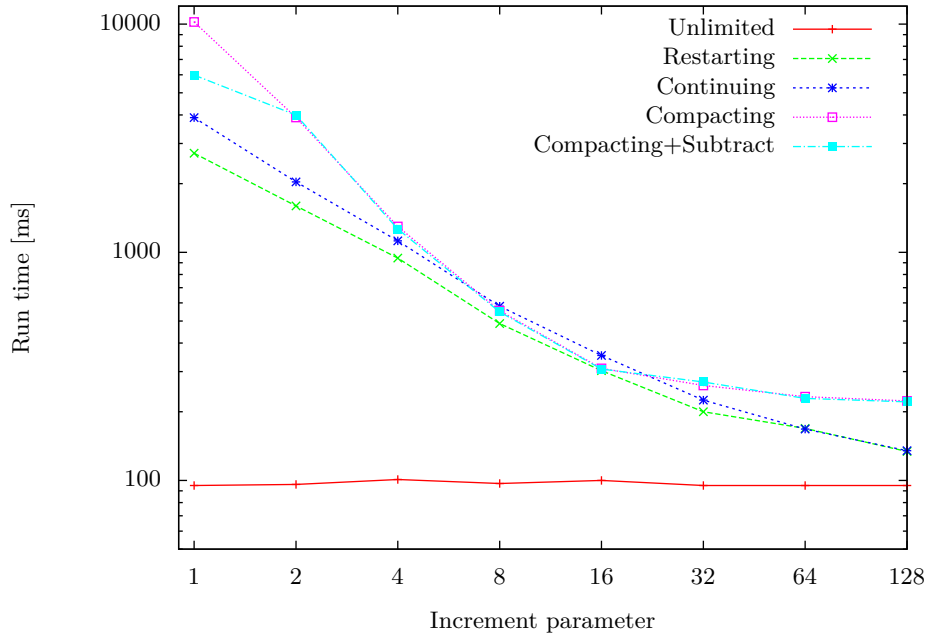


Figure 5.1: Results of full state exploration of Counter-12 model with different increment values

Consider the Counter-12 model. Multiple measurements were made on this model with different increment values. The results are shown in Table 5.2 and in Figure 5.1 (note that both axes are logarithmic on the diagram).

As it can be seen, the overhead of iterative bounded checking is significantly lower if the increment is bigger. However it has to be noted that Counter is a pathological model as discussed in Appendix B.7.

5.1.2 Evaluation of CTL expressions

It is not surprising that in most cases, the unlimited method is faster than every iterative saturation algorithm if the full state space has to be explored, because the overhead of the incremental operation can be avoided. But if the given CTL expression can be evaluated based on a (relatively small) part of the state space, the iterative methods can perform better. This is the case when the iterative algorithms have an advantage.

Table 5.3 shows the results of the measurements of evaluating various CTL expressions. Usually I used the same CTL expressions as Ciardo et al. in [12, 15, 13].

The most important observations are the following:

- If the problem is “shallow”, usually the incremental algorithms perform better. In many cases, the unlimited method was unable to calculate the result before the timeout (600 s) .

Table 5.1: *Run times of full state space exploration*

Model	Increment	Run time [s]				
		Unlim	Rest	Cont	Comp	CompSub
Counter-8	16	0.095	0.102	0.104	0.182	0.180
Counter-8	32	0.091	0.103	0.100	0.188	0.179
Counter-12	32	0.095	0.200	0.225	0.261	0.270
Counter-16	32	0.097	13.811	13.715	5.008	4.700
Counter-16	64	0.096	6.554	6.893	1.905	1.928
Hanoi-8	10	3.673	15.153	2.416	0.966	0.987
Hanoi-10	10	86.292	>600	66.940	7.391	7.445
Hanoi-12	10	>600	>600	>600	64.872	64.204
Kanban-2	10	0.095	0.108	0.107	0.211	0.202
Kanban-10	10	0.158	3.645	4.436	41.572	23.481
Phil-100	5	0.129	1.095	1.248	97.364	9.255
Queen-8	2	0.280	0.353	0.313	0.384	0.393
Queen-8	1	0.272	0.420	0.370	0.455	0.453
Queen-10	2	3.679	5.563	5.167	5.929	5.986
Queen-10	1	3.657	7.091	6.879	7.409	7.363
RR-10	10	0.269	0.577	0.351	0.524	0.464
RR-25	10	2.929	17.264	7.680	12.788	9.533
RR-50	10	24.719	361.535	117.597	223.624	118.293
SR-5	5	0.126	0.218	0.242	0.516	0.481
SR-10	5	0.286	13.191	17.346	55.509	44.462

Table 5.2: *Results of full state exploration of Counter-12 model with different increment values*

Model	Increment	Run time [s]				
		Unlim	Rest	Cont	Comp	CompSub
Counter-12	1	0.095	2.719	3.894	10.229	5.986
Counter-12	2	0.096	1.598	2.036	3.897	3.995
Counter-12	4	0.101	0.944	1.123	1.298	1.266
Counter-12	8	0.097	0.487	0.580	0.554	0.548
Counter-12	16	0.100	0.303	0.353	0.310	0.308
Counter-12	32	0.095	0.200	0.225	0.261	0.270
Counter-12	64	0.095	0.169	0.168	0.233	0.229
Counter-12	128	0.095	0.134	0.135	0.223	0.221

Table 5.3: *Run times of CTL expression evaluation*

N	Run time [s]				
	Unlim	Rest	Cont	Comp	CompSub
FMS- N (expression: $\text{EG } E[M1 > 0 \cup P1s = 3 \wedge P2s = 3 \wedge P3s = 3]$, increment: 10)					
25	2.120	40.019	40.566	18.647	21.981
50	24.501	56.307	59.514	21.964	26.284
100	396.880	61.679	63.510	21.775	26.127
1000	> 600	60.174	62.119	21.686	25.585
10,000	> 600	60.055	62.397	21.590	25.662
1,000,000	> 600	60.471	62.269	21.797	25.641
Hanoi- N (expression: $\text{EG EF } (B_8 > 0)$, increment: 10)					
12	> 600	2.489	0.810	1.548	1.536
14	> 600	2.892	0.980	2.389	1.680
16	> 600	3.253	1.110	1.829	1.817
18	> 600	3.603	1.222	1.905	1.953
20	> 600	4.079	1.425	2.137	2.098
Phil- N (expression: $E[\text{eating}_2 = 0 \cup \text{eating}_1 = 1]$, increment: 5)					
200	0.165	0.264	0.287	0.653	0.586
300	0.215	0.413	0.446	1.039	0.901
400	0.279	0.611	0.640	1.478	1.187
RR- N (expression: $\text{EG } \text{true}$, increment: 10)					
10	0.152	0.206	0.206	0.706	0.715
25	0.992	3.652	2.404	23.722	22.715
50	7.985	68.259	32.401	> 300	> 300
RR- N (expression: $E[\text{pload}_1 = 0 \cup \text{psend}_0 = 1]$, increment: 10)					
10	0.151	0.107	0.106	0.204	0.207
25	0.976	0.113	0.111	0.220	0.223
50	7.603	0.140	0.142	0.251	0.256

- What is important to see is that *there is no best solution*. According to the measurements, it is highly model and expression-dependent which bounded saturation algorithm performs the best. As it can be seen, for the Hanoi- N model with the given expression the continuing algorithm was the best. For FMS- N with the given expression, the compacting algorithm performed best.
- There are cases when large part of the state space exploration is needed to be able to evaluate the given CTL expression and the unlimited algorithm is better than the incremental algorithms. For example, this is true for the Phil- N model. For this model, the state space representation of the partial state space built by the bounded algorithms is not efficient compared to the EDD representation of the full state space.
- While there is no best solution, there were no measurements when the restarting algorithm was the best. As the aim of this thesis was to improve the performance of the restarting algorithm, it is an important observation.

5.1.3 Scalability

Scaling with the size of the model. One of the advantages of the iterative saturation-based methods that they scale with the “size of the problem” (the size of the state space fragment which is needed for CTL expression evaluation) and not with the size of the model.

Consider the checking of the $\text{EF } \text{bit}_{12} = 1$ expression on the Counter models. (This expression means the counter can reach the decimal 2^{12} value.) Using the unlimited algorithms, the full state space needs to be explored, thus the run time depends on the size of the model. However, only the same part of the model (the first 12 bits) needs to be explored to evaluate the given expression, no matter what is the size of the model. The measurements are depicted in Figure 5.2. It shows that the restarting and continuing algorithms do not depend on the size of the model. However, the run time of the compacting saturation depends on the model size. Its possible reason is that even the states that needed to be explored are the same for every model, the built state space EDDs will be different, and the depth of the recursion also depends on the size of the model.

Scaling with the size of the problem. The iterative methods perform well if the state space which is needed to be explored for the expression evaluation is relatively small. This characteristic can be observed on the Queen-10 model, by evaluating $\text{EF } (q_i = 0)$ expression (which means informally: the first i queen can be placed on the chessboard) with different i values. The measurements can be seen in Table 5.4 and in Figure 5.3.

There are two important observations. First, it can be seen that for small values of parameter i , the iterative algorithms are much better than the unlimited. But as the

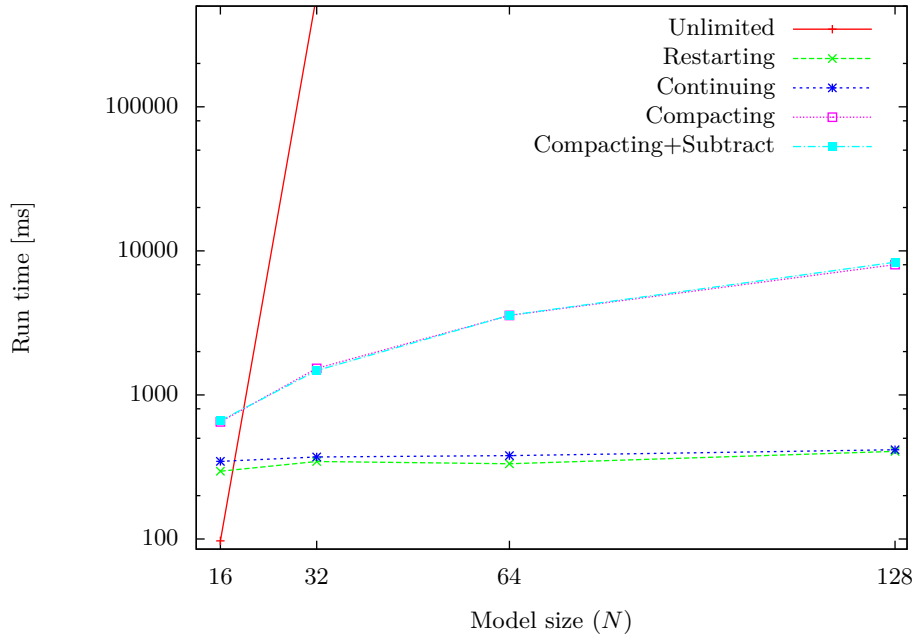


Figure 5.2: Runtime of EF $bit_{12} = 1$ evaluation on Counter- N models

Table 5.4: Results for evaluation of EF ($q_i = 0$) on Queen-10 model

Parameter i	Run time [s]				
	Unlim	Rest	Cont	Comp	CompSub
1	1.370	0.162	0.172	0.246	0.247
2	1.385	0.166	0.168	0.330	0.338
3	1.343	0.252	0.245	0.340	0.350
4	1.395	0.247	0.241	0.973	0.966
5	1.403	0.759	0.729	0.978	0.949
6	1.357	0.757	0.712	2.747	2.762
7	1.384	2.055	2.042	2.746	2.763
8	1.381	2.069	2.014	4.200	4.171
9	1.353	3.658	3.727	4.147	4.316
10	1.393	3.689	3.597	4.512	4.630

parameter i increases, the required size of the state space to be explored also increases and the advantage of the incremental algorithms decreases. If value of parameter i is high, the effect of the overhead caused by incrementality is higher than the benefit of the smaller state space exploration and the unlimited algorithm will have better results.

Another observation is that there are “plates” in the run time of incremental algorithms (i.e., the run time restarting and continuing algorithm is nearly the same for $i = 5$ and $i = 6$). Its reason is that the increment value was set to 2 in this case, so the same part of the state space is explored for $i = 5$ and $i = 6$. These plates are at different places for the restarting/continuing and the compacting algorithms, because the compacting algorithms can stop one iteration earlier in some cases, as it was discussed in Section 3.5.

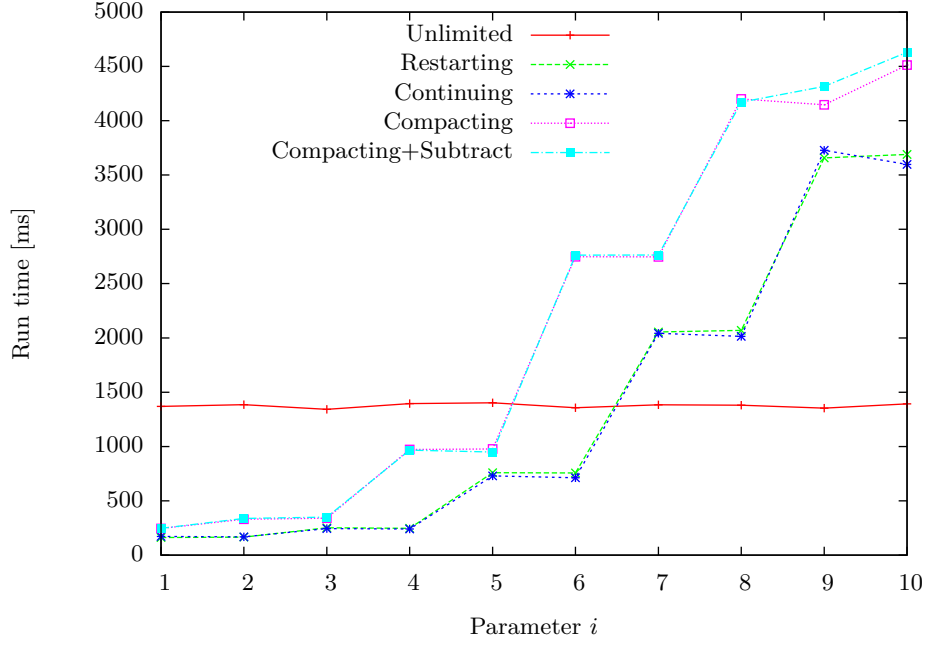


Figure 5.3: Results for evaluation of $EF (q_i = 0)$ on *Queen-10* model

Table 5.5: Measurements of peak memory consumption

Model	Expression	Incr.	Peak memory consumption (without editor) [MiB]				
			Unlim	Rest	Cont	Comp	CompSub
Hanoi-10		10	1580.92	83.11	32.43	36.25	37.41
Hanoi-12	EG $EF (B_8 > 0)$	10	> 4 GiB	86.60	27.88	38.11	32.32
Hanoi-16		10	> 4 GiB	117.73	38.84	57.18	43.85
FMS-25	EG $E[M1 > 0 \cup P1s = P2s = P3s = 3]$	10	43.86	976.56	955.50	420.53	539.24
FMS-50		10	234.66	1262.54	1165.71	471.70	631.07
FMS-100		10	1820.83	1224.42	1179.33	495.54	633.95
Queen-10	$EF (q_2 = 0)$	2	297.33	33.64	33.65	30.62	26.57
Queen-10	$EF (q_4 = 0)$	2	266.50	46.58	45.04	75.88	74.67
Queen-10	$EF (q_6 = 0)$	2	258.33	86.71	85.34	274.60	275.58
Queen-10	$EF (q_8 = 0)$	2	263.90	256.96	229.34	414.26	413.96
Queen-10	$EF (q_{10} = 0)$	2	243.85	460.58	354.12	508.95	487.47

5.2 Memory consumption measurements

This section presents memory consumption measurements of the described algorithms.

5.2.1 Peak memory consumption

Method. First, I measured the memory consumption of the PetriDotNet editor with the loaded model. Then I run a saturation algorithm and measured the peak memory consumption. The values presented below are the measured peak values of which the memory consumption of the editor is subtracted. Every metric was get by the Sysinternals Process Explorer tool where I used the “Private bytes” and “Peak private bytes” metrics.

The measurements of peak memory consumption can be seen in Table 5.5. The main observations are the following:

- One of the new iterative algorithms always requires less memory than the restarting method. Therefore it was successful to improve the memory consumption of the restarting algorithm.
- For the Hanoi model with the given expression the memory consumption of the continuing algorithm is about 30–40 % of the memory consumption of the restarting method.
- For the FMS model with the given expression the memory consumption of the compacting algorithm is about 40 % of the memory consumption of the restarting or the continuing method.
- A strange behaviour can be observed in the case of Queen–10 model using the Unlim algorithm. As we can see, while the memory consumption of the incremental algorithms is larger and larger as more and more queens have to be placed to the chessboard, the Unlim algorithm consumes less and less memory. The difference in memory consumption in the case of Unlim algorithm is from the formula checking phase. Apparently, more nodes have to be created in order to compute the expression $EF(q_2 = 0)$ than for the computation of $EF(q_{10} = 0)$ with the current decomposition of the model. It has to be noticed, that this difference between the memory consumption with different parameter values is probably much less, this difference might be due to the strategy of the garbage collector.

5.2.2 Memory allocation measurements

I also measured the number of decision-diagram-related objects created during the different analysis algorithms. For these measurements I used the CLR Profiler which provides the needed information. It has to be noted that the optimization described in Section 4.2.2 was turned on, the size of the caches was 500. The number of reused objects are not included in the measured metrics.

The measurements are shown in Table 5.6. For every algorithm, two metrics were measured: the total number of created EDD nodes and the total number of created MDD nodes. (To be precise, the constructor calls of these classes were measured.)

The EDDs are used for encoding the state space, while MDDs are used for model checking and to encode the constraints of the compacting algorithms. (MDDs are also used to encode the next-state functions, but they are not included in the number of MDD nodes as these are implemented as special MDD nodes. The number of next-state nodes are equal for every iterative methods, so their comparison is unnecessary.)

The observations concluded from the measurements are the following:

Table 5.6: *Total number of created node objects*

N	Total number of created node objects									
	Unlim		Rest		Cont		Comp		CompSub	
	EDD	MDD	EDD	MDD	EDD	MDD	EDD	MDD	EDD	MDD
Hanoi- N (expression: $\text{EG EF } (B_8 > 0)$, increment: 10)										
8	108 767	55 391	124 279	30 620	37 874	30 620	3453	80 806	3453	80 806
10	2 126 550	373 434	122 500	31 355	37 992	31 355	3508	76 634	3508	76 634
12	—	—	122 554	31 759	38 022	31 759	3560	75 777	3560	75 777
16	—	—	122 662	32 499	38 082	32 499	3664	76 421	3664	76 421
FMS- N (expression: $\text{E}[M1 > 0 \cup P1s = 3 \wedge P2s = 3 \wedge P3s = 3]$, increment: 10)										
25	17 790	91 538	770 299	214 311	774 300	214 311	205 416	622 470	315 600	622 470
50	110 506	346 688	792 204	215 378	796 302	215 378	215 237	642 396	325 148	642 396
100	783 221	1 350 113	792 224	215 378	796 322	215 378	215 237	642 396	325 148	642 396
Queen-10 (expression: $\text{EF } q_N = 0$, increment: 2)										
2	87 455	161 362	3197	630	3200	630	7392	4087	7392	4087
4	87 455	161 361	12 052	4313	10 802	4313	34 620	30 819	34 620	30 819
6	87 455	161 360	48 931	32 836	45 276	32 836	84 564	123 831	84 564	123 831
8	87 455	161 359	127 304	140 057	120 368	140 057	120 128	231 575	120 128	231 575
10	87 455	161 358	214 201	301 360	203 434	301 360	120 128	312 280	120 128	312 280

- Usually the incremental methods need less constructor calls than the unlimited algorithm.
- In the case of the Hanoi and the FMS models, where the size of the model was changed but the expression was not, the node counts of the iterative algorithms are approximately constant, while the node counts of the unlimited method is growing. Contrarily, in the case of the Queen model, where the size of the model was constant and the expression was changed, the unlimited algorithms used constant amount of nodes and the iterative algorithms used more and more nodes as the evaluation of the expression became more and more difficult.
- The MDD node count of the compacting saturation and the compacting saturation extended with subtraction is the same. That is because there is no difference between these two algorithms after the end of each iteration, exactly the same EDD will be produced by them, thus the constraints and the input of the model checker will be the same. There is a difference only in the building of the state space encoded by EDD. For similar reasons, the MDD node count of the restarting and the continuing saturation is the same too.
- We have seen earlier that the compacting algorithm performs well for Hanoi model. The possible reason can be seen on the measurements: the compacting algorithm uses about 100 times less EDD nodes.

5.3 Comparison of compacting and noncompacting methods

As it can be seen on the measurements, there are models (e.g. the full state space exploration of Hanoi- N model where compacting algorithms perform better than the unlimited

algorithm) for which the performance of the compacting algorithms is extremely good and there are models (e.g. the DPhil- N model) for which the compacting methods perform poorly. What causes this difference?

The DPhil and the Hanoi models have different characteristics: while DPhil is a rather asynchronous and symmetric model (components affect each other's behaviours only locally), Hanoi is a synchronous model. Therefore DPhil has a relatively “tight” and “dense” state space (there are lots of states at the same distance from the initial state), contrarily Hanoi has a “narrow” state space (there are states at long distance, but there are not so many states exactly at each distance). It can be easily seen in Figure 5.4 which shows the number of states by their distance values.

A difference can be observed in the size of EDDs encoding the state space too. I measured the size of state space EDDs at the end of each iteration. I also measured the initial (the “compacted”) size of the state space for the compacting method. The results can be seen in Figure 5.5. As we can see, the state space representation created by the compacting algorithm remains small for Hanoi model, but the representation of continuing method grows rapidly. The same advantage of the compacting method cannot be observed in the case of the asynchronous DPhil model.

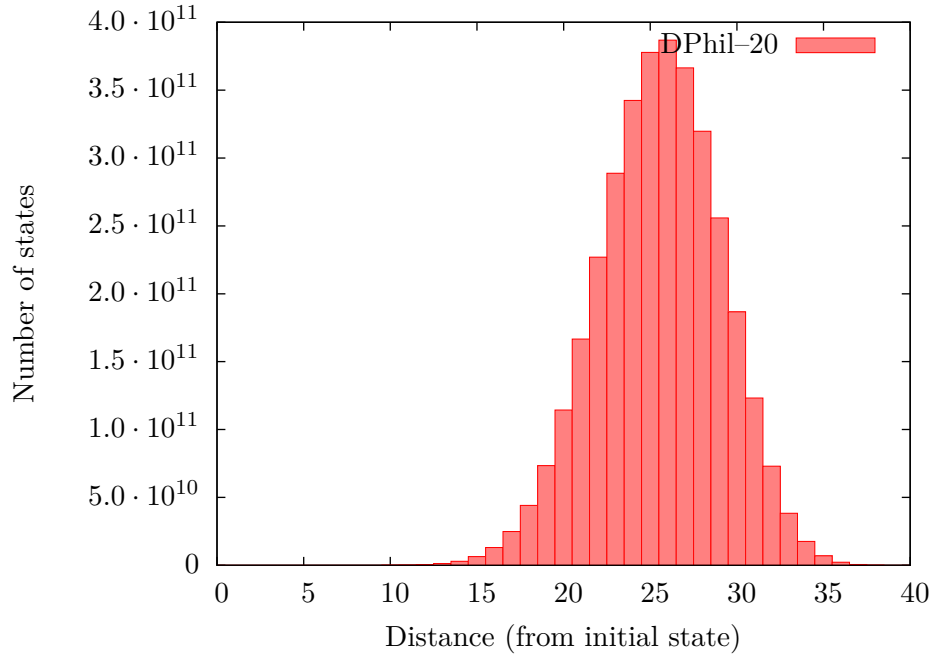
5.4 Industrial case studies

Apart from the benchmarks introduced before, I made measurements on two real cases, as shown in this section.

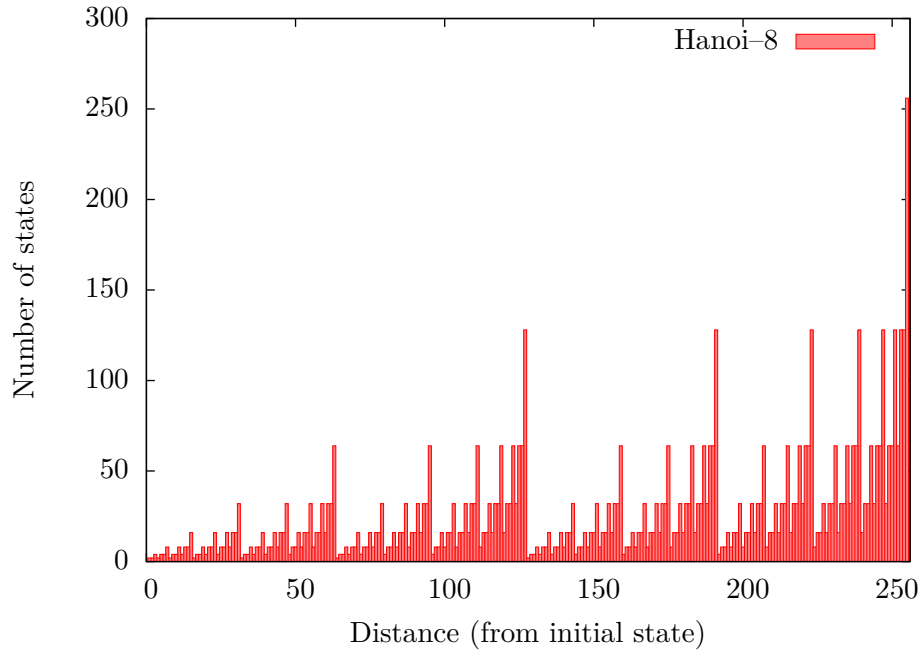
5.4.1 Verification of the PRISE logic in a nuclear power plant

In Pressurised Water Reactors (PWR), like the reactors in the Paks Nuclear Power Plant, the cooling system is divided into two circuits: a primary and a secondary coolant loop. The water in the primary circuit is directly heated by the fissile material. The turbines producing electricity take place in the secondary coolant loop where the coolant is non-contaminated. The heat is transferred between the two circuits through a heat exchanger. The damage of this fragile heat exchanger can induce the so-called *Primary to Secondary Leakage Event* (PRISE) that means the radioactive coolant from the primary loop contaminates the water in the secondary circuit. To reduce the risks of this event, a safety function is deployed that identifies the PRISE event and initiates the necessary emergency operations [42, 43].

For the measurements, I used the Coloured Petri Net (CPN) model of the PRISE safety logic. The logic was given as a Function Block Diagram (FBD) [42] as can be seen in Figure 5.7. The transformation of this FBD to Coloured Petri Net is done in [3]. The structure of the CPN model can be seen in Figure 5.8.

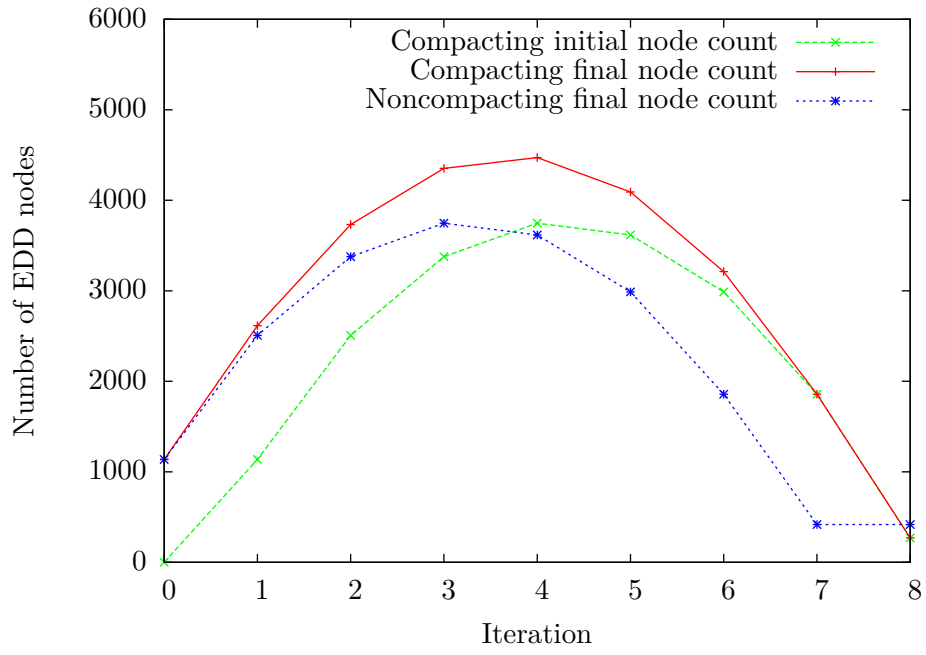


(a) Distance distribution of DPhil-20 model

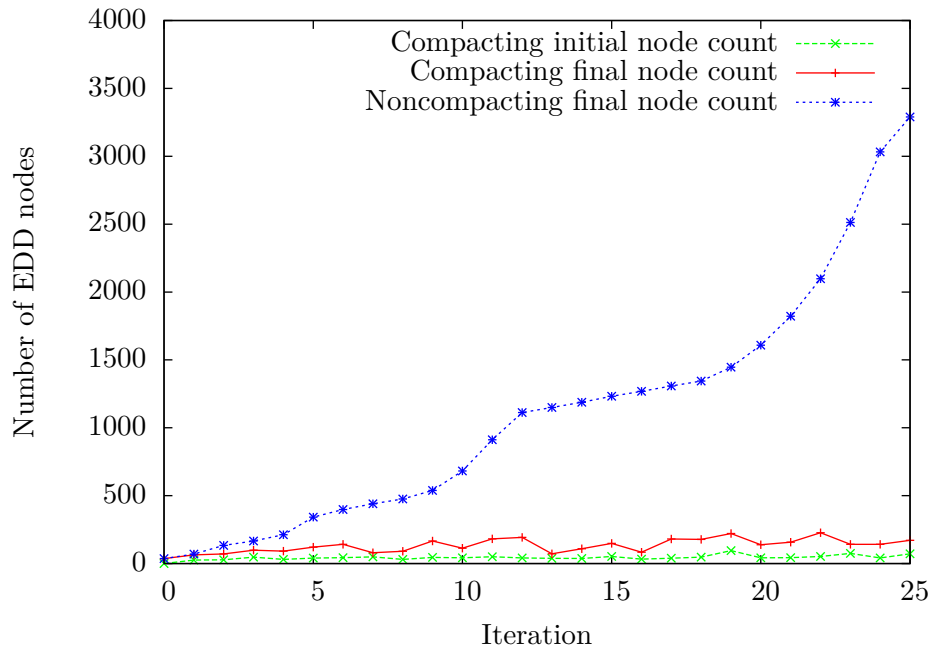


(b) Distance distribution of Hanoi-8 model

Figure 5.4: *Distance distribution of states*



(a) Size of state space EDD of DPhil-20 model



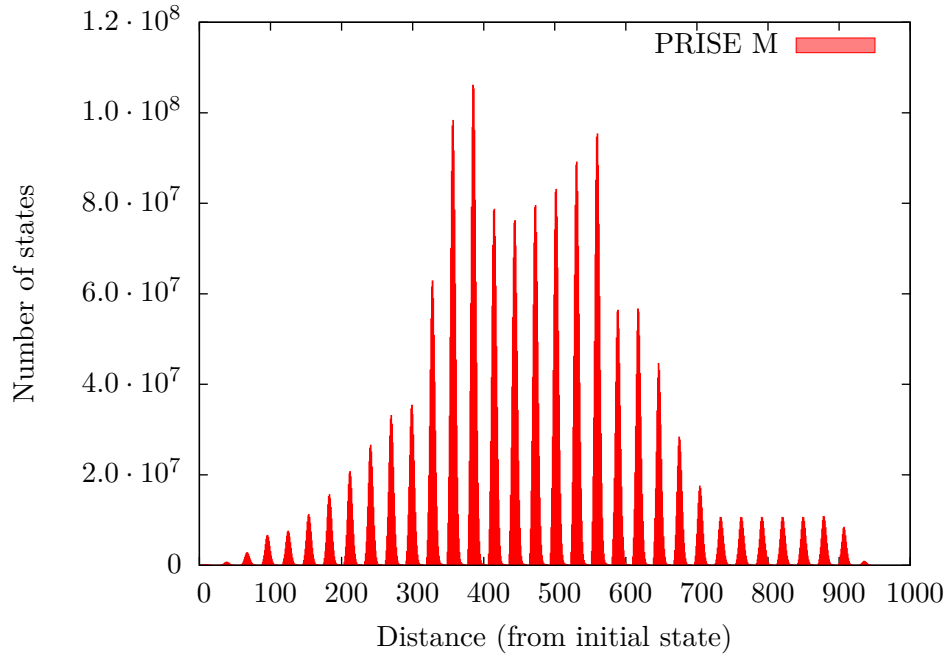
(b) Size of state space EDD of Hanoi-8 model

Figure 5.5: *Size of state space EDD*

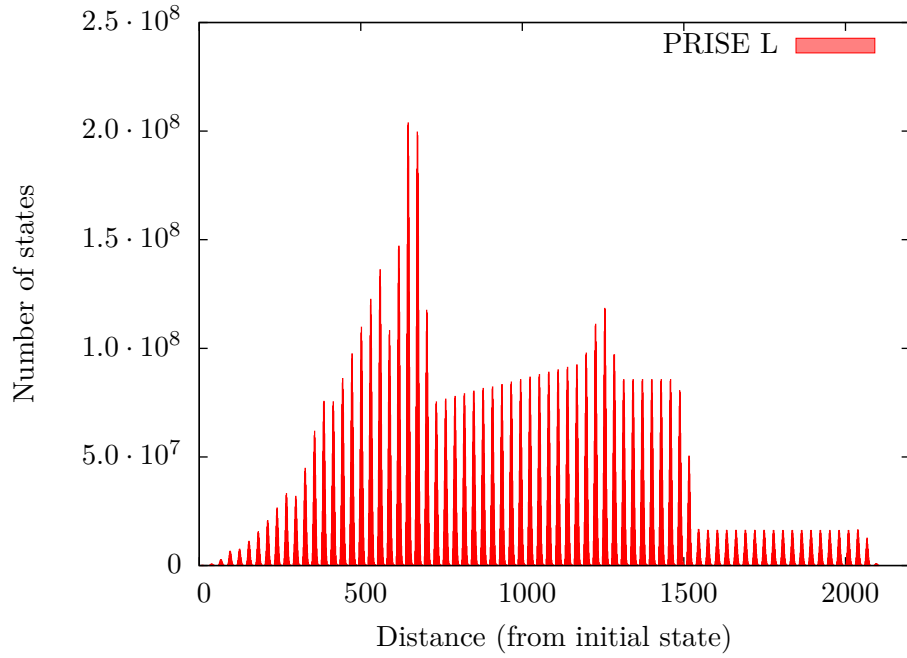
This CPN model of the PRISE safety logic is parametric: by modifying the delay of the delay module and the pulse width of the pulse modules, we can get models with different complexity. Here I consider three versions with three different size of parameters: PRISE S, PRISE M, and PRISE L. I measured the evaluation of two different expressions: the first is a simple reachability property that can be considered as relatively “shallow”, the second is a more complex, real requirement expressing that the emergency action is only initiated if it is necessary.

The run times can be seen in Table 5.7. Every measurement was performed with different initial bounds (*Init b.*) and increment values (*Incr*). Based on the results the following observations can be made:

- As the first expression refers to a relatively shallow property, the bounded model checking outperforms the non-bounded model checking, as the answer can be given based on a small part of the state space.
- For the bigger models (PRISE M and L) it can be seen that all the new bounded algorithms (continuing, compacting) are better than the original restarting algorithm. Furthermore the measurements show that the extension of the compacting saturation (CompSub algorithm) can have a huge positive impact on the model checking time.
- As it was observed previously, the bigger increment value usually means faster computation as fewer iterations are needed. However for the PRISE model, the (29,29) configuration is significantly better for all the bounded algorithms than the (40,25) configuration, even if the latter needs less iterations (see Figure 5.9). What makes the number 29 special? If we observe the distance distribution of the states in the state space (Figure 5.6) we can see a certain periodicity: the number of states at $n \cdot 29$ distance ($n \in \mathbb{N}$) is much less than at other distances. The cause is in the structure of the model. This safety logic has a cyclic behaviour, thus at the end of the cycles, the number of possible states is low. However, the order of independent computations during a cycle is not defined, thus there is a huge number of possible combinations. The compacting algorithms can highly benefit from this behaviour.
- The evaluation of the second expression relies on a big part of the state space, thus the non-bounded algorithm usually has lower run times. The restarting algorithm cannot evaluate the given CTL expression for the models PRISE M and L. The continuing algorithm provides slightly better performance. However, the compacting algorithms were able to evaluate the requirement for all the models. Moreover, using the optimal parameters, the CompSub algorithm provided even better results for the PRISE L model than the non-bounded algorithm.



(a) Distance distribution of PRISE M model

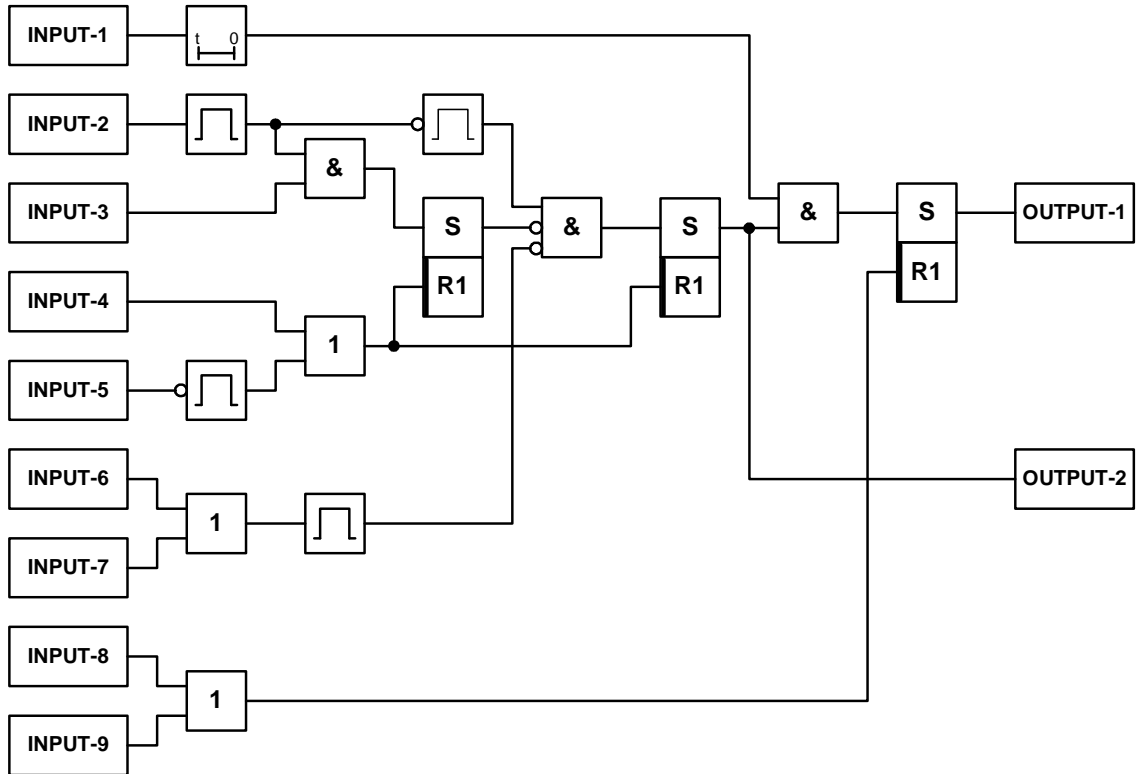


(b) Distance distribution of PRISE L model

Figure 5.6: *Distance distribution of states (PRISE model)*

Table 5.7: Run times of CTL expression evaluation on PRISE models

			Run time [s]				
Model	Init b.	Incr	Unlim	Rest	Cont	Comp	CompSub
Expression 1: $\text{EF}(\text{OUTPUT-1 can be true})$							
PRISE S	5	10	2.84	1.97	2.11	3.58	2.95
	20	20	2.76	1.52	1.56	2.42	1.81
	40	25	2.80	1.20	1.28	2.69	1.85
	29	29	2.79	1.14	1.21	1.34	1.19
PRISE M	5	10	11.18	34.13	19.58	15.02	12.26
	20	20	11.11	19.09	10.74	10.07	6.64
	40	25	11.03	14.58	9.14	11.05	6.48
	29	29	11.10	13.66	8.91	4.79	4.07
PRISE L	5	10	27.88	34.45	19.37	15.30	12.18
	20	20	28.47	19.23	10.88	10.28	6.63
	40	25	28.03	14.68	9.13	10.92	6.55
	29	29	28.42	13.72	8.98	4.80	4.02
Expression 2: there is no PRISE activation, if not necessary							
PRISE S	5	10	5.37	78.63	41.87	42.85	38.33
	29	29	5.39	33.97	19.98	12.18	10.33
PRISE M	5	10	21.96	—	—	122.62	107.97
	29	29	21.98	—	113.03	30.38	26.13
PRISE L	5	10	56.86	—	—	—	—
	29	29	56.70	—	—	51.83	46.28

**Figure 5.7:** Function Block Diagram of the PRISE safety logic [42]

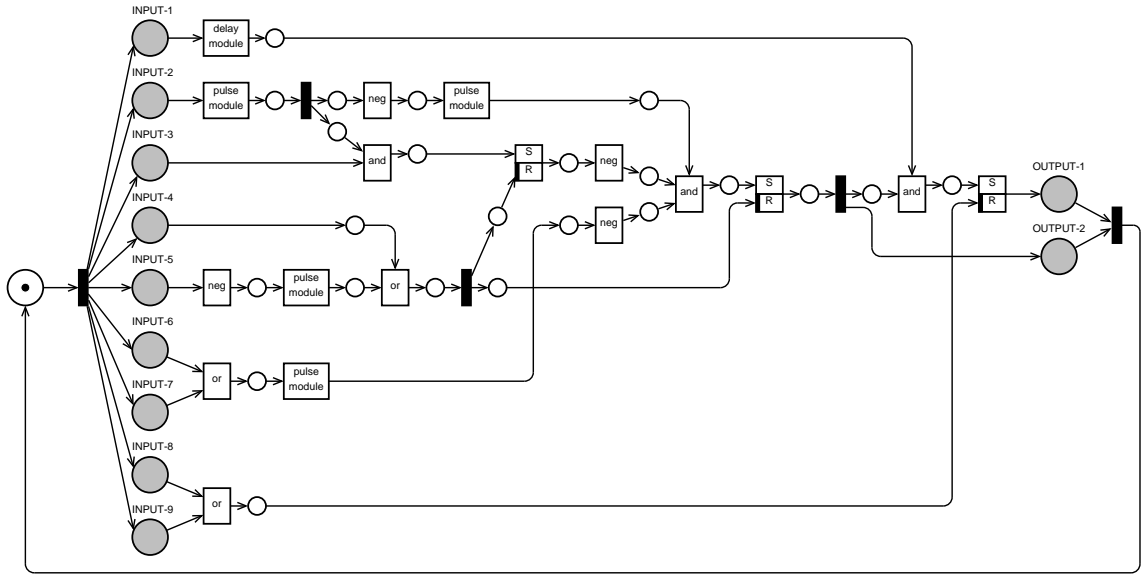


Figure 5.8: Coloured Petri Net of the PRISE safety logic [3]

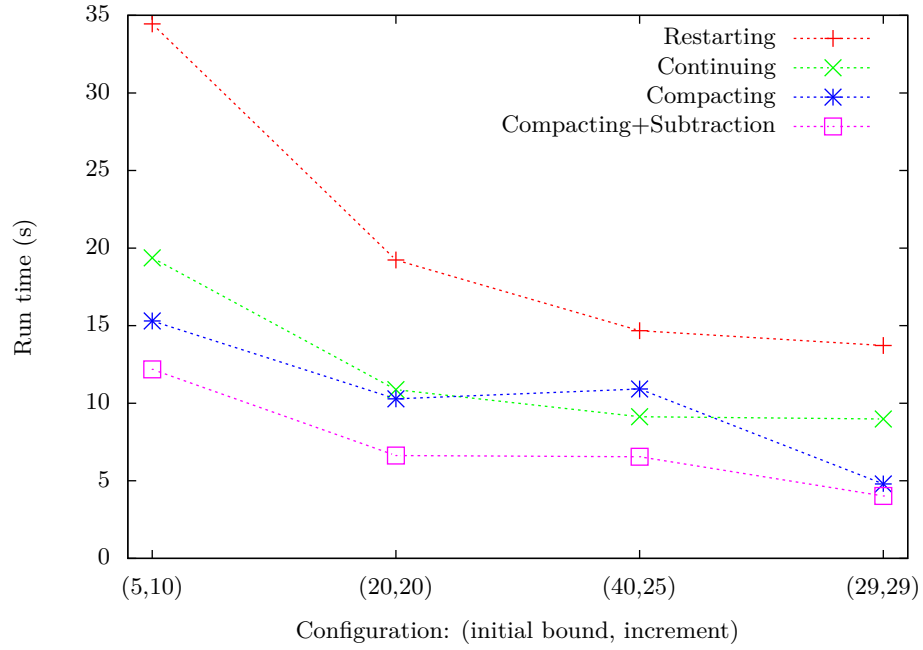


Figure 5.9: Verification time with different configurations (PRISE L)

5.4.2 Verification of PLC programs in CERN

CERN (European Organization for Nuclear Research or European Laboratory for Particle Physics) is one of the biggest scientific organizations in the world. The goal of this organization is to study and understand the fundamental structure of the universe in the frame of an international collaboration. For this reason, CERN operates numerous scientific instruments and an accelerator complex, including the Large Hadron Collider, the largest and most powerful particle accelerator on the world. The operation needs reliable auxiliary systems, such as cooling and ventilation, cryogenics, gas systems, etc. (This introduction is based on [22].)

Many of these systems use PLCs (Programmable Logic Controllers) as industrial controllers. To standardize the PLC-based systems and to facilitate the development, a framework called UNICOS (Unified Industrial Control System) is developed at CERN. It provides a standard library of common objects (called *baseline objects*) and a development method [7].

The correctness of the UNICOS baseline objects has high priority, as they are the building blocks of most PLC programs operated in CERN. Verification techniques, such as manual and automated testing were applied to the baseline objects [28, 29], but these techniques are not efficient for finding all kind of software faults, as the exhaustive testing of complex programs is typically not possible.

Model checking seems to be a good choice to complement the testing techniques in order to improve the quality of the PLC codes by finding bugs in the implementation. During my internship at CERN, I was working on applying formal verification to PLC programs. The goal was to check complex properties on the PLC code, therefore we developed a method to generate models from the PLC code automatically for the verification of properties expressed in CTL or LTL (Linear Temporal Logic) [22].

This method (see Figure 5.10) first converts the PLC programs to an internal automaton-based *intermediate model*. The CERN PLC programs are mainly written in ST language, but partially the SFC and IL languages are supported too. Then various reductions are performed on this intermediate model: simplification of the automata, merge of variables, property-dependent reductions based on the requirement to be verified, etc. After the reductions, the intermediate model is converted to the concrete syntax of a model checker tool. Currently NuSMV/nuXmv, UPPAAL, the BIP framework and PetriDotNet (or other tools supporting the PNML standard) are supported. More details can be found about the reduction techniques in [23]. A detailed case study applying this method can be found in [30].

The common intermediate model makes the method flexible and easily extendible. To include a new model checker tool, only the mapping between the intermediate model and the input format of the model checker tool have to be described, the conversion from PLC languages or the reductions are not affected by the extension.

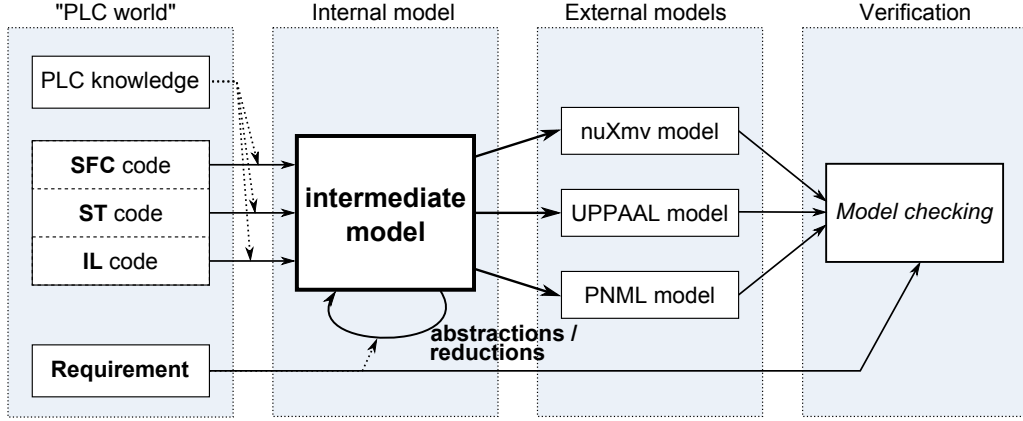


Figure 5.10: Automated PLC code modelling and verification workflow [30]

Using this method, many requirements were verified on several baseline objects. These verifications revealed many problems. Some requirements were not satisfied because they came from bad or incomplete specification. Other expressions were proved to be false because of real bugs in the implementation. These bugs were demonstrated using the real PLC hardware and software too, to show that the source of the problem is in the implementation, and not in the model generated automatically.

Our verifications were performed mainly using NuSMV/nuXmv, as its input language is close to the intermediate model and because it provided the best results so far. However for evaluation of the methods presented here, some measurements using the algorithms described in this thesis are shown. The coloured Petri net used as model is automatically generated using our tool implementing the described methodology and it benefits from the reduction techniques of the methodology.

Five requirements were used for evaluation. All of them describes requirements of the baseline object called OnOff. This is a widely-used baseline object in UNICOS whose size is representative for most of the standard objects. The requirements are the following:

- *REQ1* and *REQ2* are simple reachability properties, checking if a particular part of the code is reachable or not. Both are satisfied.
- *REQ3* and *REQ4* are real safety requirements (in the following form: $\text{AG}((\alpha \wedge \beta \wedge \dots) \rightarrow \omega)$, where the Greek letters denote Boolean predicates) given by developers. None of them are satisfied. *REQ3* is false because of a real bug in the implementation. *REQ4* is false, because its specification was not complete and it discarded a specific situation.
- *REQ5* is a real safety requirement similar to the last two, but this is satisfied.

The results of the measurements can be seen in Table 5.8. For *REQ1–REQ4*, the bounded algorithms provide better results than the unlimited algorithm. In this cases, the result can be determined based on a part of the full state space. However for *REQ5*, the whole

Table 5.8: *Run times of CTL expression evaluation on models of CERN PLC codes*

Model	Req.	Init b.	Incr	Run time [s]				
				Unlim	Rest	Cont	Comp	CompSub
OnOff	REQ1	5	5	3.18	0.14	0.14	0.20	0.20
OnOff	REQ1	10	10	3.22	0.61	0.62	0.69	0.71
OnOff	REQ2	5	5	3.21	0.99	1.07	1.15	1.15
OnOff	REQ2	10	10	3.15	1.24	1.39	1.51	1.52
OnOff	REQ3	5	5	3.15	0.99	1.08	4.74	4.48
OnOff	REQ3	10	10	3.21	1.27	1.39	2.95	3.10
OnOff	REQ4	5	5	3.19	0.98	1.08	3.58	3.57
OnOff	REQ4	10	10	3.20	1.25	1.39	2.76	2.77
OnOff	REQ5	5	5	5.81	26.07	27.83	16.43	16.45
OnOff	REQ5	10	10	5.89	17.11	17.31	11.71	11.66

state space has to be explored, therefore the bounded methods do not have benefit and the unlimited method outperforms its bounded counterpart. For these examples, typically the restarting algorithm provided the best results. However, for *REQ5*, the two compacting algorithms performed better than the restarting and compacting algorithms.

These are preliminary results but they show that bounded model checking can be used to detect shallow problems efficiently. However, if a safety requirement is satisfied, the full state space has to be explored, thus the bounded algorithms are not efficient. Consequently, the bounded methods can be useful for safety requirements in the beginning of the development process, when bugs can be found often. For mature program codes, where the occurrence of bugs is less likely, the usage of nonbounded techniques can be more efficient.

5.5 Conclusions of measurements

In this section, I compared the described bounded saturation-based algorithms by several different metrics. The conclusion was nearly the same in every case: there is no “best solution”. It is highly depending on the model and the expression to be evaluated, if the iterative methods are usable or not, and if they are usable, which iterative algorithm is the best.

It can be concluded too, that in most cases, one of the iterative algorithms proposed in this thesis perform better than the restarting method. As the goal of this thesis was to improve the performance of the restarting algorithm, the measurements proved that this goal was fulfilled.

A surprising conclusion of the measurements is that the subtracting extension does not improve the performance of the normal compacting algorithm, because usually it needs more time to execute. However it can be caused by the implementation (the saturation might be more optimized than the decision diagram operations, but the reimplementing of the diagram operations was not an objective of this work). However, in the case of full

state space exploration, it has measurable positive effects, so this extension is not useless and it is worth further analysis.

Chapter 6

Summary

This chapter summarizes the work described in this thesis. After, some future directions are drawn up.

In my thesis, I have examined the existing saturation-based algorithms, focused on the bounded state space exploration algorithms. Then I proposed new algorithms focusing to be able to perform incremental bounded state space exploration. During the evaluation it turned out that both the continuing and the compacting saturation have advantages. There are specific models and expressions to check for which the continuing saturation is better, but for other models, the compacting saturation could be a better choice. For some problems (when the problem is not “shallow”), all iterative methods perform badly, but this is obvious. Therefore the conclusion is that there is no best bounded saturation-based algorithm, the performance depends on the characteristics of the model.

All objectives of this thesis are reached:

- I have examined the existing saturation-based bounded algorithms in Section 2.6, 2.7 and 3.2.
- I have modified the existing bounded state space exploration algorithms to be able to explore states from multiple initial states. The problem is discussed in Section 3.3.1, the modified algorithm is Algorithm C.11.
- I have developed a method to ensure that the bounded state space exploration does not “reexplore” previously explored states. This method is the “negated constrained saturation” presented in Section 3.4.4.
- I have combined these solutions into a bounded saturation-based model checking algorithm, which is the compacting saturation (see Section 3.4 and Appendix C.6).
- I have demonstrated the new methods and evaluated them using measurements presented in Chapter 5.

6.1 Future work

- As there is no best bounded saturation algorithm, it is an important task to choose, which algorithm will be the most efficient. It would be useful to be able to determine it based on the characteristics of the model. Therefore a possible future direction is to develop heuristics that can analyse the Petri nets and suggest a well suited model checking method.
- The Chapter 4 showed multiple ways to improve the performance of the algorithms. However, further optimization can reduce the amount of necessary time and memory. For example, fine-tuning the garbage collector and the optimization of decision diagram library could help.
- Saturation is a general method, it can be applied for not only Petri nets. It could be interesting to port the algorithms to other formalisms (e.g., automata-based formalisms that are used for formal verification at CERN [22]) and evaluate them.

Acknowledgement

This thesis is the end (or just a milestone?) of a work started in June 2009. During these five years, my supervisors, András Vörös and dr. Tamás Bartha provided me enormous amount of help, support and knowledge. I cannot be thankful enough to them for the calm discussions, the crazy brainstormings, and the ambiance of the work in the nights, weekends and summer. Also, I would like to thank the support of the people in the Fault Tolerant Systems Research Group. Furthermore, a big thank you for the students who worked in the PetriDotNet Team together with me, especially for Attila Jámber, Vince Molnár, and Attila Klenik.

I spent my last year in CERN (European Organization for Nuclear Research). I wish to thank my colleagues for the help in the motivating work and for making this year really memorable.

In addition, I would like to thank my family for the continuous support and help.

List of Figures

1.1	Workflow of bounded model checking	15
1.2	Examples for CTL operators	18
1.3	Example Petri net	19
1.4	Encoding a binary function	21
1.5	Multivalued decision diagram	23
1.6	Edge-valued decision diagram	25
2.1	State space encoding example	29
3.1	Illustration of restarting algorithm	44
3.2	Illustration of continuing algorithm	45
3.3	Illustration of compacting algorithm	50
3.4	Example Petri net to illustrate negated constrained saturation	51
3.5	Example for compacting saturation without negated constrained saturation	52
3.6	Example for compacting saturation with negated constrained saturation . .	52
4.1	The main window of the <i>PetriDotNet</i> framework	59
4.2	Class diagrams of the decision diagram implementation	61
4.3	Number of created EDDNode objects in each iteration using different stack sizes	62
5.1	Results of full state exploration of Counter-12 model with different incre- ment values	74
5.2	Runtime of EF $bit_{12} = 1$ evaluation on Counter- N models	78
5.3	Results for evaluation of EF ($q_i = 0$) on Queen-10 model	79

5.4	Distance distribution of states	83
5.5	Size of state space EDD	84
5.6	Distance distribution of states (PRISE model)	86
5.7	Function Block Diagram of the PRISE safety logic [42]	87
5.8	Coloured Petri Net of the PRISE safety logic [3]	88
5.9	Verification time with different configurations (PRISE L)	88
5.10	Automated PLC code modelling and verification workflow [30]	90
B.1.1	One philosopher's part in Phil- N model	109
B.1.2	One philosopher's part in DPhil- N model	110
B.2.1	One node's part in SR- N model	111
B.3.1	The FMS- N model	111
B.3.2	The Kanban- N model	112
B.6.1	Part of one process in RR- N model	113
C.3.1	Public interface of the unified bounded saturation algorithm	122

List of Algorithms

3.1	Interleaved initial state creation and saturation	46
3.2	Building of initial state	46
3.3	TruncateExactEq	51
C.1	Generate (classic)	115
C.2	SatFire (classic)	115
C.3	Saturate (classic)	116
C.4	SatRecFire (classic)	116
C.5	GenerateFromInitialState (unified)	117
C.6	Generate (unified)	117
C.7	Saturate (unified)	117
C.8	SatFire (unified)	118
C.9	SatRecFire (unified)	118
C.10	BoundedGenerateFromInitialState (unified)	119
C.11	BoundedGenerate (unified)	119
C.12	BoundedSaturate (unified)	119
C.13	BoundedSatFire (unified)	120
C.14	BoundedSatRecFire (unified)	121
C.15	TruncateApprox	121
C.16	TruncateExact	121
C.17	Normalize	122
C.18	RestartingSaturation	123
C.19	ContinuingSaturation	123
C.20	CompactingSaturation	124

Bibliography

- [1] Jean-Raymond Abrial. Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, 13(5):619–628, 2007. DOI: 10.3217/jucs-013-05-0619.
- [2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [3] Tamás Bartha, András Vörös, Attila Jámbor, and Dániel Darvas. Verification of an industrial safety function using coloured Petri nets and model checking. In *Proceedings of the 14th International Conference on Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP 2012)*, pages 472–485, Budapest, Hungary, 2012. Hungarian Academy of Sciences, Computer and Automation Research Institute.
- [4] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A successful application of b in a large project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM’99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer-Verlag, 1999. DOI: 10.1007/3-540-48119-2_22.
- [5] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 164–176. ACM, 1981. DOI: 10.1145/567532.567551.
- [6] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, 2001. DOI: 10.1007/978-3-662-04558-9.
- [7] Enrique Blanco Viñuela, Jean-Michel Beckers, Benjamin Bradu, Philippe Durand, Borja Fernández Adiego, Silvia M. Izquierdo Rosas, Alexey Merezhin, Jeronimo Ortola Vidal, Jacques Rochez, and David Willeman. UNICOS evolution: CPC version 6. In *Proceedings of the 12th International Conference on Accelerator & Large Experimental Physics Control Systems ICALEPCS*, pages 786–789, Grenoble, France, 2011. <http://cds.cern.ch/record/1402490>.

- [8] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, July 1993. DOI: 10.1049/sej.1993.0025.
- [9] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. DOI: 10.1109/TC.1986.1676819.
- [10] 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):142–170, 1992. DOI: 10.1016/0890-5401(92)90017-A.
- [11] Nadia Busi. Analysis issues in Petri nets with inhibitor arcs. *Theoretical Computer Science*, 275(1–2):127–177, 2002. DOI: 10.1016/S0304-3975(01)00127-X.
- [12] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342. Springer-Verlag, 2001. DOI: 10.1007/3-540-45319-9_23.
- [13] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8:4–25, 2006. DOI: 10.1007/s10009-005-0188-7.
- [14] Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 256–273. Springer-Verlag, 2002. DOI: 10.1007/3-540-36126-X_16.
- [15] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 40–53. Springer-Verlag, 2003. DOI: 10.1007/978-3-540-45069-6_4.
- [16] Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin. Ten years of saturation: A Petri net perspective. In Kurt Jensen, Susanna Donatelli, and Jetty Kleijn, editors, *Transactions on Petri Nets and Other Models of Concurrency V*, volume 6900 of *Lecture Notes in Computer Science*, pages 51–95. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-29072-5_3.
- [17] Alessandro Cimatti, Raffaele Corvino, Armando Lazzaro, Iman Narasamdya, Tiziana Rizzo, Marco Roveri, Angela Sanseviero, and Andrei Tchaltsev. Formal verification and validation of ERTMS industrial railway train spacing system. In *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 378–393. Springer-Verlag, 2012. DOI: 10.1007/978-3-642-31424-7_29.

- [18] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2008. DOI: 10.1007/978-3-540-69850-0_1.
- [19] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1982. DOI: 10.1007/BFb0025774.
- [20] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
- [21] Dániel Darvas. Analysis of Petri net based formal models using efficient bounded model checking techniques [in Hungarian, original title: Petri-háló alapú formális modellek analízise hatékony korlátozott modellellenőrzési technikák segítségével]. BSc Thesis, Budapest University of Technology and Economics, 2011. http://petridotnet.inf.mit.bme.hu/publications/Darvas_BscThesis2011.pdf.
- [22] Dániel Darvas, Borja Fernández Adiego, and Enrique Blanco Viñuela. Transforming PLC programs into formal models for verification purposes. Internal note, CERN, 2013. CERN-ACC-NOTE-2013-0040, <http://cds.cern.ch/record/1629275/>.
- [23] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, and Víctor M. González Suárez. Formal verification of complex properties on PLC programs. In Erika Ábráham and Catuscia Palamidessi, editors, *FORTE 2014*, volume 8461 of *LNCS*, pages 284–299. Springer, 2014. To appear.
- [24] Dániel Darvas. Implementing a saturation-based model checker of asynchronous systems [in Hungarian, original title: Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez]. Report, Scientific Students’ Association, Budapest University of Technology and Economics, 2010. http://petridotnet.inf.mit.bme.hu/publications/OTDK2011_Darvas.pdf.
- [25] Dániel Darvas and Attila Jámbar. Modeling and verification of complex systems [in Hungarian, original title: Komplex rendszerek modellezése és verifikációja]. Report, Scientific Students’ Association, Budapest University of Technology and Economics, 2011. http://petridotnet.inf.mit.bme.hu/publications/TDK2011_DarvasJambor.pdf.
- [26] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971. DOI: 10.1007/BF00289519.
- [27] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer-Verlag, 1980. DOI: 10.1007/3-540-10003-2_69.

- [28] Borja Fernández Adiego, Enrique Blanco Viñuela, and Alexey Merezhin. Testing and verification of PLC code for process control. In *Proceedings of the 14th International Conference on Accelerator & Large Experimental Physics Control Systems ICALEPCS*, pages 1258–1261, San Francisco, USA, 2013. <http://cds.cern.ch/record/1697023>.
- [29] Borja Fernández Adiego, Enrique Blanco Viñuela, Jean-Charles Tournier, Víctor M. González Suárez, and Simon Bliudze. Model-based automated testing of critical PLC programs. In *11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 722–727, Bochum, Germany, 2013. IEEE. DOI: 10.1109/INDIN.2013.6622973.
- [30] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M. González Suárez. Bringing automated model checking to PLC program development – A CERN case study. In *Proceedings of the 12th IFAC–IEEE International Workshop on Discrete Event Systems*, 2014. To appear.
- [31] Limor Fix. Fifteen years of formal property verification in Intel. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, pages 139–144. Springer-Verlag, Berlin, Heidelberg, 2008. DOI: 10.1007/978-3-540-69850-0_8.
- [32] Sasha Goldshtein, Dima Zurbalev, and Ido Flatow. *Pro .NET Performance*. Apress, 2012. DOI: 10.1007/978-1-4302-4459-2.
- [33] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990. DOI: 10.1109/52.57887.
- [34] Anne Elisabeth Haxthausen. An introduction to formal methods for the development of safety-critical applications. Technical report, Technical University of Denmark, 2010. <http://www2.imm.dtu.dk/courses/02263/F13/Files/FormalMethodsNoteTS.pdf>.
- [35] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel® Core™ i7 processor execution engine validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer-Verlag, 2009. DOI: 10.1007/978-3-642-02658-4_32.
- [36] Ralf Kneuper. Limits of formal methods. *Formal Aspects of Computing*, 9(4):379–394, 1997. DOI: 10.1007/BF01211297.
- [37] Thierry Lecomte, Thierry Servat, and Guilhem Pouzancre. Formal methods in safety-critical railway systems. In *Proceedings of the 10th Brazilian Symposium on Formal Methods (SBMF)*, Ouro Preto, Brazil, 2007.

- [38] John J. Marciniak. *Encyclopedia of Software Engineering*, volume 1. John Wiley & Sons, Inc., 1994. DOI: 10.1002/0471028959.
- [39] Roland Meyer, Johannes Faber, Jochen Hoenicke, and Andrey Rybalchenko. Model checking duration calculus: A practical approach. *Formal Aspects of Computing*, 20(4–5):481–505, 2008. DOI: 10.1007/s00165-008-0082-7.
- [40] Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*, pages 6–25. Springer-Verlag, 1999. DOI: 10.1007/3-540-48745-X_2.
- [41] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. DOI: 10.1109/5.24143.
- [42] Erzsébet Németh and Tamás Bartha. Formal verification of safety functions by reinterpretation of functional block based specifications. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2009. DOI: 10.1007/978-3-642-03240-0_17.
- [43] 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 and System Safety*, 94(5):942–953, 2009. DOI: 10.1016/j.ress.2008.10.012.
- [44] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri net analysis using boolean manipulation. In *Application and Theory of Petri Nets 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, 1994. DOI: 10.1007/3-540-58152-9_23.
- [45] Radek Pelánek. BEEM: Benchmarks for explicit model checkers. In *Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer-Verlag, 2007. DOI: 10.1007/978-3-540-73370-6_17.
- [46] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 1982. DOI: 10.1007/3-540-11494-7_22.
- [47] Radu Siminiceanu and Gianfranco Ciardo. *Symbolic Model Checking for Avionics*, pages 85–112. John Wiley & Sons, Inc., 2012. DOI: 10.1002/9781118459898.ch5.
- [48] Bertalan Szilvási. Development of an education tool for the formal methods course [in Hungarian, original title: Oktatási segédeszköz fejlesztése Formális módszerek tárgyhöz]. MSc Thesis, Budapest University of Technology and Economics, 2008.

- [49] András Vörös and András Pataricza. Multiple valued decision diagrams in the diagnosis of IT systems. Technical report, BUTE DMIS – IBM FA research project, 2009.
- [50] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation based CTL model checking. In Jaan Penjam, editor, *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST’11*, pages 149–160, Tallinn, Estonia, 2011. Tallinn University of Technology, Institute of Cybernetics. http://petridotnet.inf.mit.bme.hu/publications/SPLST2011_VorosDarvasBartha.pdf.
- [51] András Vörös, Dániel Darvas, and Tamás Bartha. Bounded saturation-based CTL model checking. *Proceedings of the Estonian Academy of Sciences*, 62(1):59–70, 2013. DOI: 10.3176/proc.2013.1.07.
- [52] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):19:1–19:36, October 2009. DOI: 10.1145/1592434.1592436.
- [53] Andy Jinqing Yu, Gianfranco Ciardo, and Gerald Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *International Journal on Software Tools for Technology Transfer*, 11:117–131, 2009. DOI: 10.1007/s10009-009-0099-0.
- [54] Yang Zhao and Gianfranco Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 2009. DOI: 10.1007/978-3-642-04761-9_27.
- [55] Homepage of the *PetriDotNet* framework. (accessed: 15 March 2014) <http://petridotnet.inf.mit.bme.hu/en/>.

Appendix A

Abbreviations and used symbols

A.1 Symbols

$\mathbf{0}$	terminal zero node of a (binary or multivalued) decision diagram
$\mathbf{1}$	terminal one node of a (binary or multivalued) decision diagram
\perp	terminal node in an EDD
α	event in a discrete-state model ($\alpha \in \mathcal{E}$)
$\mathcal{A}(v)$	above operator (the set of tuples encoded from the root node to node v)
$\mathcal{B}(v)$	below operator (the set of tuples encoded from node v to the terminal node)
$Bot(\alpha)$	bottom function (the lowest level on which event α is not independent)
$\delta(\mathbf{s})$	distance of the state \mathbf{s} from the initial state
D_i	the domain of the i th level in a decision diagram
E	set of edges in a Petri net
\mathcal{E}	set of events
\mathcal{E}_k	events whose <i>Top</i> value is k
$\mathbf{i} = (i_K, \dots, i_1)$	global states in a partitioned model
K	number of submodels
$level(n)$	level number of node n in a decision diagram
$M(p)$	marking of the place p in a Petri net (number of tokens on place p)
\mathcal{N}	next-state function/relation (set of possible state-state transitions)
\mathcal{N}_α	next-state function of event α
\mathbb{N}	set of nonnegative integer numbers: $\{0, 1, 2, \dots\}$
P	set of places in a Petri net
r	root node of a decision diagram
ρ	weight of the dangling edge in an EDD
$\langle s, w \rangle$	an edge in an EDD which points to the node w with label $s \in \mathbb{N} \cup \{\infty\}$

s_0	initial state of a model
\mathcal{S}	set of reachable states
$\hat{\mathcal{S}}$	potential state space
\mathcal{S}^{init}	set of initial states
\mathcal{S}_i	local state space of the i th submodel
$\mathcal{S}_{iter=i}$	state space explored during iteration i
$\mathcal{S}_{b,k}$	k -bounded part of reachable states (states reachable from the initial state within $\leq k$ transitions)
$supp(\alpha)$	support set of event α (levels on which event α is not independent)
T	set of transitions in a Petri net
$Top(\alpha)$	top function (the highest level on which event α is not independent)
V	vertex set of a graph
$value(n)$	value of terminal node n in a binary or multivalued decision diagram (0 for terminal zero node, 1 for terminal one node)
$w(e)$	weight of the edge e in a Petri net
x_i	the i th represented variable of a decision diagram
\mathbb{Z}^+	set of positive integer numbers: $\{1, 2, \dots\}$

A.2 Abbreviations

BDD Binary Decision Diagram

CERN European Organization for Nuclear Research or European Laboratory for Particle Physics

CPN Coloured Petri net

CPU Central Processing Unit

CTL Computation Tree Logic

DAG Directed Acyclic Graph

DPhil- N Dining philosophers model with N philosophers (which can contain deadlock)

EDD (or EV^+ MDD) Edge-valued Decision Diagram

FBD Function Block Diagram

JIT compiler Just-In-Time compiler

LINQ Language Integrated Query (part of .NET framework)

MDD Multivalued (Multiway) Decision Diagram

Phil- N Dining philosophers model with N philosophers (which cannot contain deadlock)

PLC Programmable Logic Controller

PN Petri net

PRISE Primary to Secondary Leakage Event

QMDD Quasi-reduced Multivalued (Multiway) Decision Diagram

RAM Random Access Memory

RR- N Round robin protocol with N participants

SR- N Slotted ring protocol with N participants

UNICOS Unified Industrial Control System

Appendix B

Models

This chapter introduces the models used for evaluation in Chapter 5. Every discussed model can be downloaded from the PetriDotNet website [55].

B.1 Dining philosophers

The *Dining philosophers* model is the generalization of the well-known *Five Dining Philosophers* problem [26]. In this problem five philosophers sit around a circular table. Each philosopher has a plate with full of spaghetti. There are five forks on the table, one between each two plates. To eat the dish, each philosopher needs two available forks aside their plate. Therefore two neighbours cannot eat at the same time.

Every philosopher can eat or think, but not at the same time. If a philosopher gets hungry and wants to start to eat, he must get two forks. After he finished to eat, he will put the forks back to the table immediately (but not before).

As the reader can see, there is a chance for getting into a deadlock. If all philosophers get hungry at the same time and everyone gets the left fork first, every philosopher will have only one fork, there will not be any more forks on the table. With one fork nobody can eat their dish, so everyone remains hungry and nobody will put the forks back to the table.

For evaluation purposes, I used two variants of this problem:

- a simplified variant without deadlocks (Phil- N), and
- a normal variant with possible deadlock (DPhil- N).

B.1.1 Phil- N model

This model is based on the Five Dining Philosophers problem, but instead of five philosophers, there are N philosophers around the table. Another modification is that the philoso-

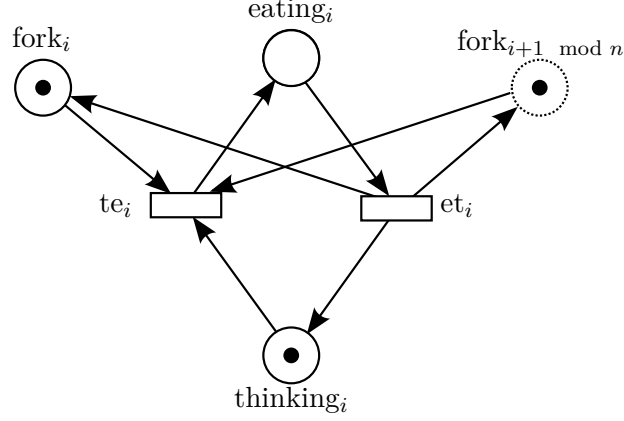


Figure B.1.1: One philosopher's part in *Phil-N* model

phers get both forks at the same time, therefore they cannot hold single forks in their hand. This modification eliminates the chance of deadlock.

The Petri net model of *Phil-N* can be seen on Fig. B.1.1. This model is easy to understand. If $fork_i$ place is marked, then the i th fork is on the table. If $thinking_i$ is marked, then the i th philosopher is thinking. Otherwise $eating_i$ is marked and the i th philosopher is eating (with his two forks).

Usual partitioning for the saturation-based algorithms: every philosopher forms 1-1 partition.

B.1.2 DPhil-N model

This model is the straightforward model of the generalized Dining Philosophers problem with N philosophers. The Petri net model of *DPhil-N* can be seen on Fig. B.1.2.

If the i th philosopher is thinking, the $Thinking_i$ place is marked. If he gets hungry, the $GoEat_i$ transition will fire and he will wait for two forks. If he gets the left fork, the $HasLeft_i$ place will be marked. If both $HasLeft_i$ and $HasRight_i$ are marked, the philosopher can eat. After that, the $Release_i$ transition will fire, the philosopher puts back the forks and returns to thinking.

Usual partitioning for the saturation-based algorithms: every philosopher forms 2-2 partition (3 place / partition).

B.2 Slotted ring protocol

The *Slotted ring protocol* model with N participants (*SR-N*) models a computer network in which nodes are aligned into a circle, so all node has exactly two neighbours. The communication uses fixed size frames which can be empty or occupied (used). [40, 44]

The Petri net model of *SR-N* can be seen in Figure B.2.1. If an empty frame arrives to the i th node from the $(i + 1 \bmod N)$ th node, the $Free_{i+1 \bmod N}$ transition fires. If

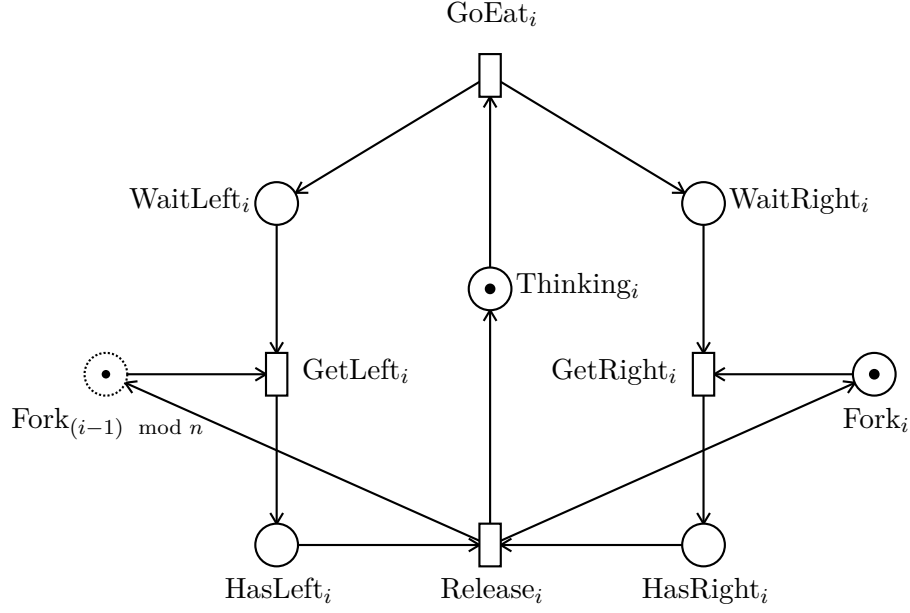


Figure B.1.2: One philosopher's part in *DPhil-N* model

the arriving frame is used, the $Used_{i+1 \bmod N}$ transition fires. The marking of each place means the following:

C_i	The i th node is ready to receive a frame from node $i + 1 \bmod N$.
A_i	The i th node received an occupied (used) frame.
B_i	The i th node received a free frame.
D_i	The i th node is ready to process the free frame that will be sent.
H_i	The i th node is ready to process the occupied frame that will be sent.
F_i	The i th node is ready to send a frame to node $i - 1 \bmod N$.
E_i	The i th node is ready to send a free frame.
G_i	The i th node is ready to send an occupied frame.

Usual partitioning for the saturation-based algorithms: every node forms 2-2 partition (4 place / partition).

B.3 Manufacturing systems

The *FMS* model represents a flexible manufacturing system as it is described in [16]. There are multiple type of parts to assemble and there are N of each part. The model can be seen in Figure B.3.1.

A similar model is the *Kanban* model [16] which models an assembly line using Kanban scheduling system. The model can be seen in Figure B.3.2.

For both models, the structure is the same for every value N , only the number of tokens depends on the parameter.

Usual partitioning for the saturation-based algorithms: every place is in separate partition.

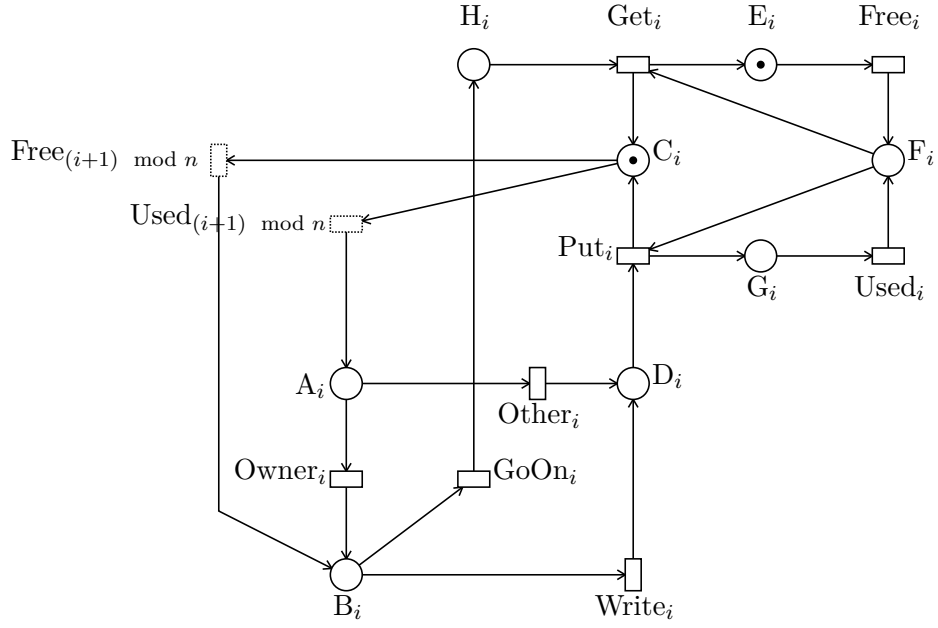


Figure B.2.1: One node's part in $SR-N$ model

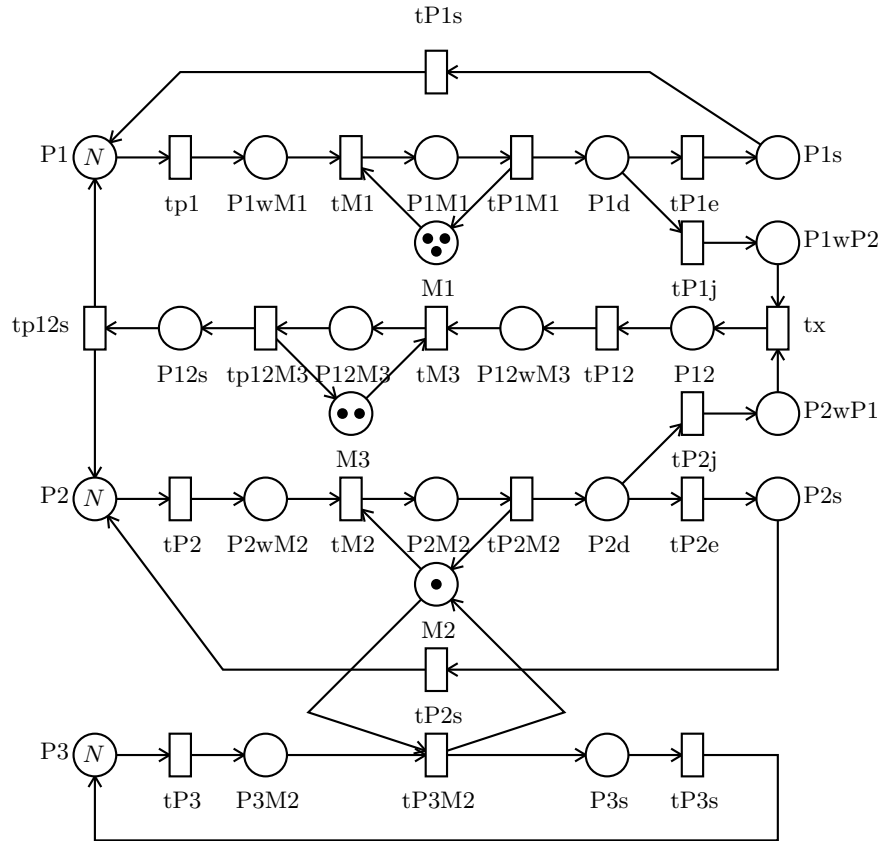


Figure B.3.1: The $FMS-N$ model

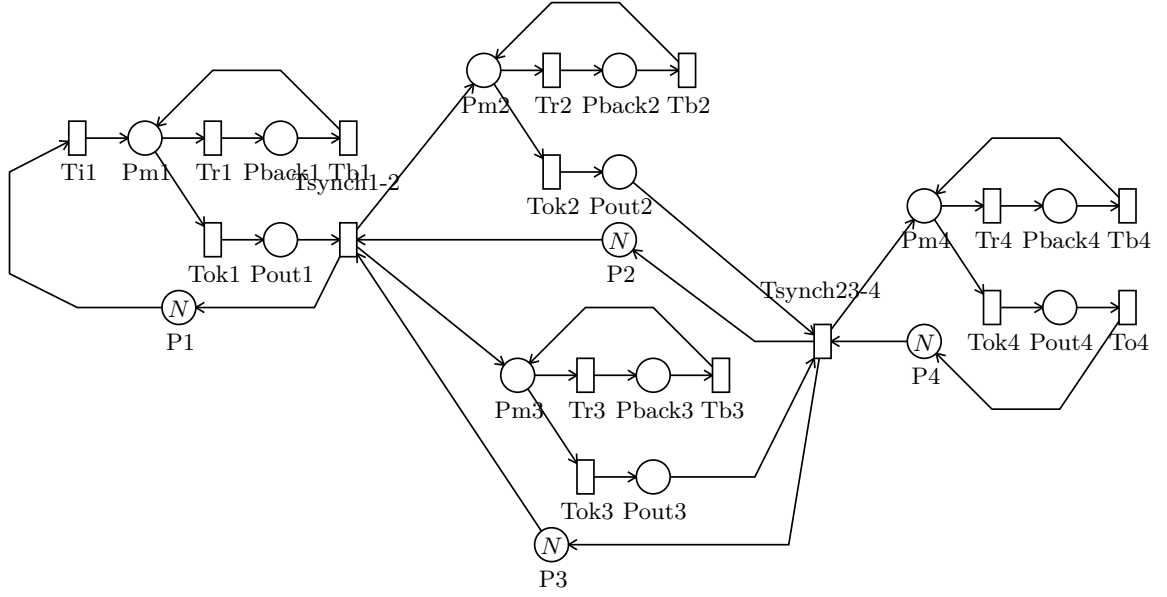


Figure B.3.2: *The Kanban- N model*

B.4 Tower of Hanoi

The *Tower of Hanoi* (or Hanoi- N for short) model is the Petri net model of the well-known Tower of Hanoi game. There are three rods (A , B , C) and N disks on the rod A . Each disk has different size: the 0th disk is the smallest, the $(N - 1)$ th disk is the biggest. The disks can only be placed on top of smaller rings, only the topmost disk can be moved from one rod to another and only one disk can be moved at a time. The goal of the game is to move all disks from rod A to rod C [45].

The Petri net model contains places $P = \{A_0, A_1, \dots, A_{N-1}, B_0, \dots, B_{N-1}, C_0, \dots, C_{N-1}\}$. If there is a token in the A_i place it means the i th disk is on the A rod. For all $x, y \in \{A, B, C\}$ and for all $i \in \{0, 1, \dots, N-1\}, j \in \{\text{empty}, i, i+1, \dots, N-1\}$ there is a transition $t_{x,y,i,j}$ which means the i th disk is moved from rod x to rod y while the largest disk on y is the j th. In the model there is an edge from x_i to $t_{x,y,i,j}$ and from $t_{x,y,i,j}$ to y_i which corresponds to the transfer of the disk. The other constraints (there is no smaller disk on x or on y than i) are expressed with inhibitor edges. The model contains large amount of elements (for example, Hanoi-8 contains 222 transitions and 2040 edges).

Usual partitioning for the saturation-based algorithms: every disk forms 1-1 partition (3 place / partition).

B.5 Queen

The *Queen- N* model models the famous N -queens problem. The goal is to place N queens on an $N \times N$ chessboard. For this problem we used the model presented in [16]. The model contains $N^2 + N$ places: $p_{i,j} \forall i, j \in 1, \dots, N$ places signify that there is a queen on the (i, j) position. There are also $q_i \forall i \in 1, \dots, N$ places. If there is a token in the q_i place, it

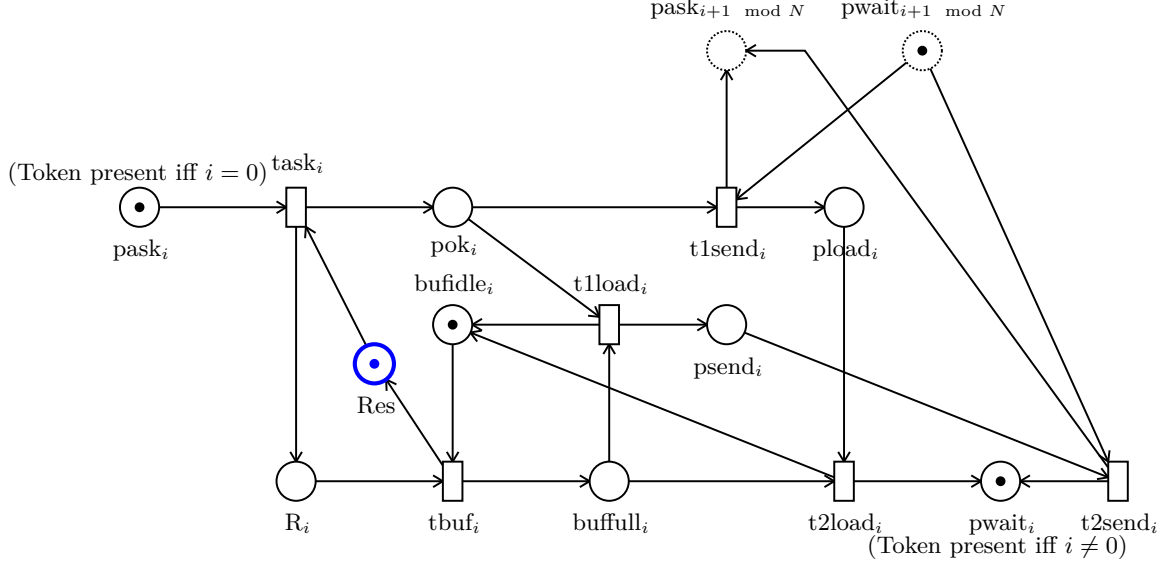


Figure B.6.1: Part of one process in $RR-N$ model

means there is no queen in the i th row at the moment. The queens fill the table in order: the first queen will be placed to the first row, the second queen to the second row, etc. The other constraints are expressed by inhibitor edges.

According to [16], this model is “pathologically difficult for symbolic methods”.

Usual partitioning for the saturation-based algorithms: every place forms separate partition.

B.6 Round robin

The round robin model ($RR-N$) models a “round-robin resource sharing protocol, where N processes cyclically can access a resource” [16]. The model of one process can be seen in Figure B.6.1. Note that the place Res is global (there is only one place Res for all processes).

Usual partitioning for the saturation-based algorithms: every process forms 2-2 partition (4 place / partition), and place Res is in a separate partition.

B.7 Counter

The Counter- N model models an N -bit binary counter counting from 0 to $2^N - 1$ [16]. In this model, there are N places (bit_0, \dots, bit_{N-1}) representing each bit of the actual counter value. There are also $N + 1$ transitions: inc_0, \dots, inc_{N-1} and $reset$. The firing of transition inc_i removes one-one token from bit_0, \dots, bit_{i-1} places and puts a token into bit_i place. The transition inc_i is disabled, if there is a token in place bit_i which is expressed by an inhibitor edge. The $reset$ transition removes one-one token from every place bit_i .

It has to be noticed that Counter model is a pathological model. Usually the graph of states is branching. As it can be seen in Figure 5.4, in the case of DPhil-20 model, there are 20 possible state from the initial state. 230 states are reachable by two firing from the initial state and there are more than $6.3 \cdot 10^7$ states at 10 distance. Contrarily, in the Counter- N model every global state has exactly one successor state, so the graph of states is topologically equivalent to a N -length circle which is quite unusual.

Usual partitioning for the saturation-based algorithms: every place forms separate partition.

Appendix C

Pseudocodes

C.1 General unbounded saturation

Algorithm C.1: Generate (classic)

```

    output : MDDNode
1  MDDNode last  $\leftarrow$  1;
2  for  $i = 1$  to topLevelNumber do
3      MDDNode  $n \leftarrow$  NewNode( $i$ );
4       $n[0] \leftarrow$  last;
5      last  $\leftarrow$   $n$ ;
6      Saturate(last);
7   $s \leftarrow$  CheckIn( $s$ );
8  return  $s$ ;

```

Algorithm C.2: SatFire (classic)

```

    input   :  $\alpha$ : event,  $q$ : MDDNode,  $r$ : MDDNode
    output  : changed: bool

1  int  $l \leftarrow q.level$ ;
2  bool changed  $\leftarrow$  false;
3   $L \leftarrow$  LocalStateTransitionsToExplore( $r$ ); //  $L = \{(i, j) : r[i][j] \neq 0\}$ 
4  foreach  $(i, j) \in L$  do
5      MDDNode  $f \leftarrow$  SatRecFire( $\alpha, q[i], r[i][j]$ );
6      if  $f = 0$  then continue ;
7      MDDNode  $u \leftarrow$  Union( $f, q[j]$ );
8      if  $u \neq q[j]$  then
9          if  $j \notin S_l$  then Confirm( $l, j$ );
10          $q[j] \leftarrow u$ ;
11         changed  $\leftarrow$  true;
12         if  $r[j] \neq 0$  then
13              $L \leftarrow L \cup \{(j, x) \in \text{LocalStateTransitionsToExplore}(r)\}$ ;
14  return changed;

```

Algorithm C.3: Saturate (classic)

```
input    :  $p$ : MDDNode
1 bool  $changed \leftarrow false$ ;
2 while  $changed$  do
3   |  $changed \leftarrow false$ ;
4   | foreach  $e \in \mathcal{E}_k$  do
5   |   |  $changed \leftarrow \text{SatFire}(\alpha, p, \mathcal{N}_\alpha) \vee changed$ ;
```

Algorithm C.4: SatRecFire (classic)

```
input    :  $\alpha$ : event,  $q$ : MDDNode,  $r$ : MDDNode
output   : MDDNode

1 int  $l \leftarrow q.level$ ;
2 if CacheFind( $FIRE$ ,  $(\alpha, q)$ , out  $s$ ) then return  $s$ ;
3 MDDNode  $s \leftarrow \text{NewNode}(l)$ ;
4 bool  $changed \leftarrow false$ ;
5  $L \leftarrow \text{LocalStateTransitionsToExplore}(r)$ ; //  $L = \{(i, j) : r[i][j] \neq 0\}$ 
6 foreach  $(i, j) \in L$  do
7   | MDDNode  $f \leftarrow \text{SatRecFire}(\alpha, q[i], r[i][j])$ ;
8   | if  $f = 0$  then continue;
9   | MDDNode  $u \leftarrow \text{Union}(f, s[j])$ ;
10  | if  $u \neq s[j]$  then
11  |   | if  $j \notin S_l$  then Confirm( $l, j$ );
12  |   |  $s[j] \leftarrow u$ ;
13  |   |  $changed \leftarrow true$ ;
14 if  $changed$  then Saturate( $s$ );
15  $s \leftarrow \text{CheckIn}(s)$ ;
16 PutIntoCache( $FIRE$ ,  $(\alpha, q)$ ,  $s$ );
17 return  $s$ ;
```

C.2 Unified unbounded saturation

Algorithm C.5: GenerateFromInitialState (unified)

```

input   : cons: MDDNode, negCons: MDDNode
output  : MDDNode
1 // cons: root node of the constraint
2 // negCons: root node of the negated constraint
3 MDDNode last  $\leftarrow$  1;
4 for  $i = 1$  to topLevelNumber do
5   MDDNode  $n \leftarrow$  NewNode( $i$ );
6    $n[0] \leftarrow last$ ;
7    $last \leftarrow n$ ;
8 return Generate(last, cons, negCons);

```

Algorithm C.6: Generate (unified)

```

input   :  $s$ : MDDNode, cons: MDDNode, negCons: MDDNode
output  : MDDNode
1 //  $s$  : root node of initial state space
2 // cons: root node of the ponated constraint (set of states can be included)
3 // negCons: root node of the negated constraint (set of states must be omitted)
4 if Generate( $s$ , cons, negCons) was already called then return;
5 if level( $s$ )  $\geq 1$  then
6   for  $i = 0$  to  $D_{level(s)}$  do
7     if  $s[i] = 0$  then continue;
8     Generate( $s[i]$ , cons[ $i$ ], negCons[ $i$ ]);
9 Saturate( $s$ , cons, negCons);
10  $s \leftarrow$  CheckIn( $s$ );
11 return  $s$ ;

```

Algorithm C.7: Saturate (unified)

```

input   :  $p$ : MDDNode, cons: MDDNode, negCons: MDDNode
1 bool changed  $\leftarrow$  true;
2 while changed do
3   changed  $\leftarrow$  false;
4   foreach  $e \in \mathcal{E}_k$  do
5     changed  $\leftarrow$  SatFire( $\alpha, p, \mathcal{N}_\alpha$ , cons, negCons)  $\vee$  changed;

```

Algorithm C.8: SatFire (unified)

```

input   :  $\alpha$ : event,  $q$ : MDDNode,  $r$ : MDDNode,  $cons$ : MDDNode,  $negCons$ : MDDNode
output  :  $changed$ : bool

1  $l \leftarrow q.level$ ;
2  $changed \leftarrow false$ ;
3  $L \leftarrow LocalStateTransitionsToExplore(r)$ ; //  $L = \{(i, j) : r[i][j] \neq 0\}$ 
4 foreach  $(i, j) \in L$  do
5   if  $cons[j] = 0$  then continue;
6   if  $negCons[j] \neq 0 \wedge l = 1$  then continue;
7   MDDNode  $f \leftarrow SatRecFire(\alpha, q[i], r[i][j], cons[j], negCons[j])$ ;
8   if  $f = 0$  then continue;
9   MDDNode  $u \leftarrow Union(f, q[j])$ ;
10  if  $u \neq q[j]$  then
11    if  $j \notin S_l$  then Confirm( $l, j$ );
12     $q[j] \leftarrow u$ ;
13     $changed \leftarrow true$ ;
14    if  $r[j] \neq 0$  then
15      // if there is potential state transition from state j
16       $L \leftarrow L \cup \{(j, x) : (j, x) \in LocalStateTransitionsToExplore(r)\}$ ;
17 return  $changed$ ;

```

Algorithm C.9: SatRecFire (unified)

```

input   :  $\alpha$ : event,  $q$ : MDDNode,  $r$ : MDDNode,  $cons$ : MDDNode,  $negCons$ : MDDNode
output  : MDDNode

1  $l \leftarrow q.level$ ;
2 if  $cons = 1 \wedge l < Bot(e)$  then return  $q$ ;
3 if  $CacheFind(FIRE, (\alpha, q, cons, negCons), out\ s)$  then return  $s$ ;
4 MDDNode  $s \leftarrow NewNode(l)$ ;
5  $changed \leftarrow false$ ;
6  $L \leftarrow LocalStateTransitionsToExplore(r)$ ; //  $L = \{(i, j) : r[i][j] \neq 0\}$ 
7 foreach  $(i, j) \in L$  do
8   if  $cons[j] = 0$  then continue;
9   if  $negCons[j] \neq 0 \wedge l = 1$  then continue;
10  MDDNode  $f \leftarrow SatRecFire(\alpha, q[i], r[i][j], cons[j], negCons[j])$ ;
11  if  $f = 0$  then continue;
12  MDDNode  $u \leftarrow Union(f, s[j])$ ;
13  if  $u \neq s[j]$  then
14    if  $j \notin S_l$  then Confirm( $l, j$ );
15     $s[j] \leftarrow u$ ;
16     $changed \leftarrow true$ ;
17 if  $changed$  then Saturate( $cons, negCons$ );
18  $s \leftarrow CheckIn(s)$ ;
19  $PutIntoCache(FIRE, (\alpha, q, cons, negCons), s)$ ;
20 return  $s$ ;

```

C.3 Unified bounded saturation

These pseudocodes are based on our earlier work [51] and on [50, 53]. The improvements are marked with colours: the extensions due to the constrained saturation are marked with **blue**, the negated constrained extensions are in **red**. Note that the constrained extensions are not used in the iterative algorithms, they are here just for the completeness of the algorithms.

Algorithm C.10: BoundedGenerateFromInitialState (unified)

```

input   : cons: MDDNode, negCons: MDDNode
output  : EDDEdge
1 // cons: root node of the constraint
2 // negCons: root node of the negated constraint
3 EDDNode last  $\leftarrow \perp$ ;
4 for  $i = 1$  to TopLevelNumber do
5   EDDNode  $n \leftarrow \text{NewNode}(i)$ ;
6    $n[0] \leftarrow \langle 0, \text{last} \rangle$ ;
7   last  $\leftarrow n$ ;
8 return  $\text{Generate}(\langle 0, \text{last} \rangle, \text{cons}, \text{negCons})$ ;

```

Algorithm C.11: BoundedGenerate (unified)

```

input   :  $\langle v, s \rangle$ : EDDEdge, cons: MDDNode, negCons: MDDNode
output  : EDDEdge
1 //  $\langle v, s \rangle$  : root edge of initial state space
2 // cons: root node of the ponated constraint (set of states can be included)
3 // negCons: root node of the negated constraint (set of states must be omitted)
4 if  $\text{CacheFind}(\text{GEN}, (\langle v, s \rangle, \text{cons}, \text{negCons}), \text{out } a)$  then return  $\langle a, s \rangle$ ;
5 if  $\text{level}(s) \geq 1$  then
6   for  $i = 0$  to  $D_{\text{level}(s)}$  do
7     if  $s[i].\text{label} = \infty$  then continue;
8      $\text{Generate}(s[i], \text{cons}[i], \text{negCons}[i])$ ; // return value is intentionally not used
9  $\text{BoundedSaturate}(\langle v, s \rangle, \text{cons}, \text{negCons})$ ;
10  $\text{int } \gamma \leftarrow \text{Normalize}(s)$ ;
11  $s \leftarrow \text{CheckIn}(s)$ ;
12  $\text{PutIntoCache}(\text{GEN}, (\langle v, s \rangle, \text{cons}, \text{negCons}), \gamma + v)$ ;
13 return  $\langle \gamma + v, s \rangle$ ;

```

Algorithm C.12: BoundedSaturate (unified)

```

input   :  $\langle v, p \rangle$ : EDDEdge, cons: MDDNode, negCons: MDDNode
1 bool changed  $\leftarrow \text{true}$ ;
2 while changed do
3   changed  $\leftarrow \text{false}$ ;
4   foreach  $e \in \mathcal{E}_k$  do
5     changed  $\leftarrow \text{BoundedSatFire}(\alpha, \langle v, p \rangle, \mathcal{N}_\alpha, \text{cons}, \text{negCons}) \vee \text{changed}$ ;

```

C.3.1 Bounded saturation as a module

As all iterative bounded saturation algorithm uses the basic saturation-based algorithms, it is easier to think about the basic algorithms as reusable modules, as it can be seen on

Algorithm C.13: BoundedSatFire (unified)

```
input   :  $\alpha$ : event,  $\langle v, q \rangle$ : EDDEdge,  $r$ : MDDNode, cons: MDDNode, negCons: MDDNode
output  : changed: bool

1 int  $l \leftarrow q.level$ ;
2 bool changed  $\leftarrow false$ ;
3  $L \leftarrow LocalStateTransitionsToExplore(r)$ ; //  $L = \{(i, j) : r[i][j] \neq 0\}$ 
4 foreach  $(i, j) \in L$  do
5   if  $p[i].label \geq bound$  then continue;
6   if  $cons[j] = 0$  then continue;
7   if  $negCons[j] \neq 0 \wedge l = 1$  then continue;
8    $\langle y, f \rangle \leftarrow BoundedSatRecFire(\alpha, q[i], r[i][j], cons[j], negCons[j])$ ;
9    $\langle w, s \rangle \leftarrow Truncate(\langle y + 1, f \rangle)$ ;
10  if  $w = \infty$  then continue;
11   $\langle u, p \rangle \leftarrow UnionMin(\langle w, s \rangle, q[j])$ ;
12  if  $\langle u, p \rangle \neq q[j]$  then
13    if  $j \notin S_i$  then Confirm( $l, j$ );
14     $q[j] \leftarrow \langle u, p \rangle$ ;
15    changed  $\leftarrow true$ ;
16    if  $r[j] \neq 0$  then
17      // if there is potential state transition from state j
18       $L \leftarrow L \cup \{(j, x) \in LocalStateTransitionsToExplore(r)\}$ ;
19 return changed;
```

Figure C.3.1. In this way, the parameter passing is easier to understand.

- The *Truncate* field sets the used truncating method (exact or approximate).
- The *bound* field sets the used bound value.
- The *initial* field stores the root edge of the initial state space encoded by an EDD.
- The *stateSpace* field stores the root edge of the state space (explored by the last execution) encoded by an EDD.
- The *constraint* field stores the root node of the constraint encoded by an MDD.
- The *negConstraint* field stores the root node of the negated constraint encoded by an MDD.
- The *topLevelNumber* field stores the number of the root level of the decision diagrams.
- The *BoundedSaturation* method starts the state space exploration using the parameters set by the fields and stores the result in the state space field. (Basically, it performs the following: $stateSpace \leftarrow \text{Generate}(initial, constraint, negConstraint)$.)

Algorithm C.14: BoundedSatRecFire (unified)

```

input   :  $\alpha$  : event,  $\langle v, q \rangle$  : EDDEdge,  $r$  : MDDNode, cons : MDDNode, negCons : MDDNode
output  : EDDEdge

1 int  $l \leftarrow q.level$ ;
2 if negCons = 0  $\wedge$  cons = 1  $\wedge$   $l < Bot(e)$  then return  $\langle v, q \rangle$ ;
3 if CacheFind(FIRE, ( $\alpha, q, cons, negCons$ ), out  $\langle \alpha + v, s \rangle$ ) then return  $\langle \alpha + v, s \rangle$ ;
4 // extension with subtraction (only present in CompSub algorithm)
5 if  $l < Bot(e)$  then
6   | EDDNode  $\langle rv, rn \rangle \leftarrow SubtractMDD(q, negCons)$ ;
7   | PutIntoCache(FIRE, ( $\alpha, q, cons, negCons$ ),  $\langle rv - v, rn \rangle$ );
8   | return  $\langle rv, rn \rangle$ ;
9 EDDNode  $s \leftarrow NewNode(l)$ ;
10 bool changed  $\leftarrow false$ ;
11  $L \leftarrow LocalStateTransitionsToExplore(r)$ ; //  $L = \{(i, j) : r[i][j] \neq 0\}$ 
12 foreach  $(i, j) \in L$  do
13   | if cons[ $j$ ] = 0 then continue;
14   | if negCons[ $j$ ]  $\neq$  0  $\wedge$   $l = 1$  then continue;
15   |  $\langle y, f \rangle \leftarrow BoundedSatRecFire(\alpha, q[i], r[i][j], cons[j], negCons[j])$ ;
16   |  $\langle w, s \rangle \leftarrow Truncate(\langle y, f \rangle)$ ;
17   | if  $w = \infty$  then continue;
18   |  $\langle u, p \rangle \leftarrow UnionMin(\langle w, s \rangle, s[j])$ ;
19   | if  $\langle u, p \rangle \neq s[j]$  then
20     | if  $j \notin \mathcal{S}_i$  then Confirm( $l, j$ );
21     |  $s[j] \leftarrow \langle u, p \rangle$ ;
22     | changed  $\leftarrow true$ ;
23 if changed then
24   | BoundedSaturate( $s, cons, negCons$ );
25 int  $\gamma \leftarrow Normalize(s)$ ;
26 EDDNode  $s \leftarrow CheckIn(s)$ ;
27 PutIntoCache(FIRE, ( $\alpha, q, cons, negCons$ ),  $\langle \gamma, s \rangle$ );
28 return  $\langle \gamma + v, s \rangle$ ;

```

Algorithm C.15: TruncateApprox

```

input   :  $\langle v, p \rangle$  : EDDEdge
output  : EDDEdge

1 if  $v > bound$  then
2   | return  $\langle \infty, \perp \rangle$ ;
3 else
4   | return  $\langle v, p \rangle$ ;

```

Algorithm C.16: TruncateExact

```

input   :  $\langle v, p \rangle$  : EDDEdge
output  : EDDEdge

1 if  $v > bound$  then
2   | return  $\langle \infty, \perp \rangle$ ;
3 if  $p.level = 0$  then
4   | return  $\langle v, p \rangle$ ;
5 if CacheFind(TRUNC,  $\langle v, p \rangle$ , out  $t$ ) then
6   | return  $\langle v, t \rangle$ ;
7 EDDNode  $n \leftarrow NewNode(p.level)$ ;
8 foreach  $i \in S_{p.level}$  do
9   |  $r \leftarrow TruncateExact(\langle v + p[i].label, p[i].node \rangle)$ ;
10  |  $n[i] \leftarrow \langle r.label - v, r.node \rangle$ ;
11  $n \leftarrow CheckIn(n)$ ;
12 PutIntoCache(TRUNC,  $\langle v, p \rangle, n$ );
13 return  $\langle v, n \rangle$ ;

```

Algorithm C.17: Normalize

```

input   :  $s$  : EDDNode
output  : int
1 int  $\gamma \leftarrow \min\{s[i].label \mid i \in D_{level(s)}\};$ 
2 foreach  $i \in D_{level(s)}$  do
3   |  $s[i].label \leftarrow s[i].label - \gamma;$ 
4 return  $\gamma;$ 

```

BoundedSaturationData
+ Truncate : TruncateMethod
+ bound : int
+ initial : EDDEdge
+ stateSpace : EDDEdge
+ constraint : MDDNode
+ negConstraint : MDDNode
+ topLevelNumber : int {readonly}
+ BoundedSaturation()

Figure C.3.1: *Public interface of the unified bounded saturation algorithm*

C.4 Restarting bounded saturation

Algorithm C.18: RestartingSaturation

```
input   : incr : int
output  : result of model checking

1 sd ← new BoundedSaturationData;
2 sd.Truncate ← TruncateExact;           // sets the truncate method
3 sd.bound ← incr;
4 int i ← 1;                             // i: iteration counter
5 while true do
6   sd.bound ← i · incr;
7   // The sd.initial is empty, so saturation will start from the initial state of
   the Petri net.
8   sd.BoundedSaturation();
9   ModelChecking();
10  if model checking was able to determine the result then
11    | return result of model checking;
12  i ← i + 1;
```

C.5 Continuing bounded saturation

As it can be seen, there is only small difference between the restarting saturation (Algorithm C.18) and the continuing saturation (Algorithm C.19). But to be able to implement the continuing saturation, the base bounded algorithms had to be extended, because restarting saturation needed a special case only (when the initial state set contains only one state).

Algorithm C.19: ContinuingSaturation

```
input   : incr : int
output  : result of model checking

1 sd ← new BoundedSaturationData;
2 sd.Truncate ← TruncateExact;           // sets the truncate method
3 int i ← 1;                             // i: iteration counter
4 while true do
5   sd.bound ← i · incr;
6   sd.BoundedSaturation();
7   ModelChecking();
8   if model checking was able to determine the result then
9     | return result of model checking;
10  // Continue the next iteration from the state space of this iteration.
11  sd.initial ← sd.stateSpace;
12  i ← i + 1;
```

C.6 Compacting bounded saturation

Algorithm C.20: CompactingSaturation

```
input   : incr : int
output  : result of model checking

1 sd ← new BoundedSaturationData;
2 sd.Truncate ← TruncateExact;           // sets the truncate method
3 sd.bound ← incr;
4 int i ← 0;                               // i: iteration counter
5 MDDNode negConstraint ← 0;              // root of the negated constraint MDD
6 EDDEdge initialEDD;
7 while true do
8   if i > 0 then
9     | sd.negConstraint ← negConstraint;
10    | sd.initial ← initialEDD;
11    | sd.BoundedSaturation();
12    | ModelChecking();
13    if model checking was able to determine the result then
14      | return result of model checking;
15    else
16      | negConstraint ← Union(negConstraint, sd.stateSpace.AsMDD());
17      | initialEDD ← TruncateExactEq(sd.stateSpace, incr); // copies states that are
      | exactly at incr distance
```

Appendix D

Details of measurements

D.1 Measurements of implementation details

Listing D.1.1: *Measurement for array indexing runtime.*

```
1  const long ITER = 100000000;  
2  
3  private static void MeasA()  
4  {  
5      Stopwatch stopwatch = Stopwatch.StartNew();  
6      int[] array = new int[1000];  
7      int x;  
8      for (long i = 0; i < ITER; i++)  
9      {  
10         x = array[0];    // compilation optimizations should be disabled!  
11     }  
12     stopwatch.Stop();  
13     Console.WriteLine("Measurement: {0} ms; {1} ms/iter", stopwatch.ElapsedMilliseconds, (double)↵  
14         stopwatch.ElapsedMilliseconds / ITER);  
15 }
```

Listing D.1.2: *Measurement for array.First() runtime.*

```
1  const long ITER = 100000000;  
2  
3  private static void MeasB()  
4  {  
5      Stopwatch stopwatch = Stopwatch.StartNew();  
6      int[] array = new int[1000];  
7      int x;  
8      for (long i = 0; i < ITER; i++)  
9      {  
10         x = array.First();  
11     }  
12     stopwatch.Stop();  
13     Console.WriteLine("Measurement: {0} ms; {1} ms/iter", stopwatch.ElapsedMilliseconds, (double)↵  
14         stopwatch.ElapsedMilliseconds / ITER);  
15 }
```

Listing D.1.3: *Measurement for overhead of delegates.*

```
1 public static class DelegateTest
2 {
3     private const long ITER = 100000000;
4
5     [System.Runtime.CompilerServices.MethodImpl
6         (System.Runtime.CompilerServices.MethodImplOptions.NoInlining)]
7     public static void TestMethod() { }
8
9     public delegate void VoidDelegate();
10
11     public static void Run()
12     {
13         MeasDirectCall();
14         MeasDelegate(TestMethod);
15     }
16
17     public static void MeasDirectCall()
18     {
19         Stopwatch stopwatch = Stopwatch.StartNew();
20         for (long i = 0; i < ITER; i++)
21         {
22             TestMethod();
23         }
24         stopwatch.Stop();
25         Console.WriteLine("Measurement direct call: {0} ms; {1} ms/iter",
26             stopwatch.ElapsedMilliseconds,
27             (double)stopwatch.ElapsedMilliseconds / ITER);
28     }
29
30     public static void MeasDelegate(VoidDelegate del)
31     {
32         Stopwatch stopwatch = Stopwatch.StartNew();
33         for (long i = 0; i < ITER; i++)
34         {
35             del();
36         }
37         stopwatch.Stop();
38         Console.WriteLine("Measurement delegate call: {0} ms; {1} ms/iter",
39             stopwatch.ElapsedMilliseconds,
40             (double)stopwatch.ElapsedMilliseconds / ITER);
41     }
42 }
```

D.2 Measurement tools

- CLR Profiler

Available at <http://www.microsoft.com/en-us/download/details.aspx?id=16273>. This tool provides rich information about the memory allocation and consumption of managed programs.

- Windows Software Development Kit (SDK) for Windows 8

Available at <http://msdn.microsoft.com/en-us/windows/desktop/hh852363.aspx>. It contains WinDbg which is a powerful low-level debugging tool.

- Windows Performance Toolkit

Part of the Windows Assessment and Deployment Kit (ADK) available at <http://www.microsoft.com/en-us/download/details.aspx?id=30652>. It includes two powerful tools: Windows Performance Recorder and Windows Performance Analyzer.

- Sysinternals Process Explorer This tool provides rich information about running programs. It is available at <http://www.sysinternals.com>.

D.3 Computer used for the measurements

The configuration of the computer used for measuring is the following:

- processor: Intel® Core™ i5 660 (3.3 GHz)
- memory: 8 GiB
- operating system: Microsoft Windows 7 Enterprise (64 bit edition)
- framework: .NET 4.5 Framework
- high-resolution performance counter is supported, frequency: 3,247,119 tick/s
- PetriDotNet version 1.3.4853.20859