# DSSZ-MC – A Tool for Symbolic Analysis of Extended Petri Nets

Monika Heiner, Martin Schwarick, and Alexej Tovchigrechko

Computing Science Institute, Brandenburg University of Technology
Postbox 10 13 44, 03013 Cottbus, Germany
dsszmc@informatik.tu-cottbus.de

**Abstract.** DSSZ-MC supports the symbolic analysis of bounded place/transition Petri nets extended by read, inhibitor, equal, and reset arcs. No previous knowledge of the precise boundedness degree is required. It contains tools for the efficient analysis of standard properties (boundedness, liveness, reversibility) and CTL model checking, built on an object-oriented implementation of Zero-suppressed Binary Decision Diagrams and Interval Decision Diagrams. The main features are saturation-based state space generation, analysis of strongly connected components, dead state analysis with trace generation, and CTL model checking by limited backward reachability analysis. The tool is available for Windows, Linux, and Mac/OS.

## 1 Motivation

Considering efficient implementations of model checking tools for Petri nets, most research efforts so far are aimed at techniques for 1-bounded nets.

Reports on applying symbolic techniques to k-bounded nets can be occasionally found in the literature, but the only available tool implementing symbolic CTL model checking is SMART [CS03]. Though every bounded net can be simulated by a 1-bounded net, in practice the transformation is generally complicated and produces huge nets which can no longer be efficiently analysed.

However, biochemical networks in systems and synthetic biology often result in k-bounded models, caused by stoichiometry and the number of molecules or discrete concentration levels involved; see e.g. [KJH05], [GH06], [HGD08]. Many of these models could not be analysed by existing model checking tools and new techniques were required.

This paper gives an overview on the basic functionality of our new symbolic analysis tool that supports the efficient analysis of 1-bounded (zbdd-mc) and k-bounded (idd-mc) Petri nets extended by four non-standard arc types. It replaces our former symbolic CTL model checker of 1-bounded place/transition Petri nets [Noa99], which has been part of the model checking kit [SSE03] for quite a while.

We deliberately confine ourselves to an informal presentation; see [Tov08] for more detailed information concerning data structures and algorithms, as well as all related formal definitions.

## 2 Inputs

### 2.1 Extended Petri Nets

The tool supports the analysis of standard place/transition Petri nets extended for convenience by four special arc types: read arcs (identified by a black dot), inhibitor arcs (hollow dot), equal arcs (two black dots), and reset arcs (double arrow), which can be used simultaneously and always go from places to transitions. The standard firing rule is adapted accordingly. The enabling condition is extended in the following way: if there is an arc $a$ with a weight $w = V(p,t)$ connecting a place $p$ with a transition $t$, then $t$ can be enabled in a marking $m$ if the following conditions are satisfied:

- $a$ is a read arc $\wedge\, m(p) \geq w$,
- $a$ is an inhibitor arc $\wedge\, m(p) < w$,
- $a$ is an equal arc $\wedge\, m(p) = w$.

The token situation on $p$ is not changed by the firing of $t$, i.e. $m'(p) = m(p)$. Contrary, reset arcs do not alter the enabling condition, but involve a change of the marking on $p$ by firing of $t$:

- $m'(p) = 0$, if $p$ is not also a postplace of $t$
- $m'(p) = V(t,p)$, if $p$ is also a postplace of $t$

This net class is strictly more powerful than the class of standard place/transition Petri nets. Non-standard arcs help to express context conditions and allow themselves an elegant implementation of the analysis algorithms. For illustration consider the extended Petri nets in Figure 1, showing typical components of software verification models.

In the tradition of our previous model checking tools, the Petri nets have to be given in the Abstract Petri Net Notation (APNN) [BKK94] which is a language for the description of different classes of Petri nets. Keywords are similar to LaTeX commands. APNN has been adapted to allow the specification of the non-standard arc types. The input format can be generated, e.g., by the export feature of our hierarchical Petri net editor Snoopy [HRS08] which supports standard place/transition Petri nets as well as the extended Petri net class.

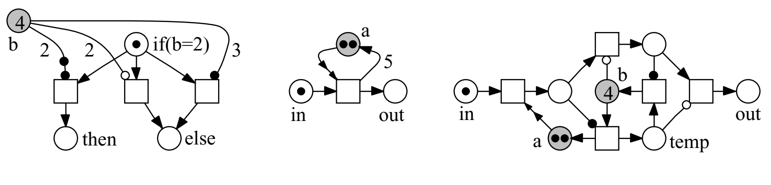


**Fig. 1.** Extended Petri nets components for software modelling (from left to right): if (b=2) then ... else ..., a := 5, a := b

All analyses supported by DSSZ-MC involve the construction of the state space, so the models have to be bounded. However, no previous knowledge of the exact boundedness degree is required. Please note, the tool does not perform a coverability check. If the Petri net is unbounded, a run will not terminate before the memory is exhausted (physically or as set by the memory option).

## 2.2 CTL Model Checking

We support model checking of Computational Tree Logic (CTL) according to the standard semantics; for a formal semantics definition see e.g. [CGP01]. The following grammar specifies a valid input for our CTL model checker.

```
ctl_input          = ctl_formula ';'
                   | ctl_formula ';' ctl_input .
ctl_formula        = '(' ctl_formula ')'
                   | unop '(' ctl_formula')'
                   | '('ctl_formula')' binop '('ctl_formula')'
                   | ae '[' ctl_formula 'U' ctl_formula ']'
                   | untemp '(' ctl_formula ')'
                   | ap .
unop               = '!' .
binop              = '*' | '+' | '->' | '<-' | '<->' .
ae                 = 'A' | 'E' .
untemp             = 'AX' | 'EX' | 'AF' | 'EF' | 'AG' | 'EG' .
(* when using zbdd-mc *)
ap                 = PLACE .
------------------------------------------------------------------
(* when using idd-mc *)
ap                 = PLACE cmp num
                   | PLACE 'in' interval .
cmp                = '==' | '!=' | '>=' | '>' | '<=' | '<' .
interval           = '[' num ',' num ')' .
num                = [0-9]+ .
```

PLACE has to be a valid place name of the Petri net to be analysed and in conformity with the standard conventions of $C^{++}$ identifiers. Places are read as Boolean variables in the case of 1-bounded Petri nets (zbdd-mc) and as integer variables in the case of k-bounded Petri nets (idd-mc).

Intervals are left-closed and right-open; thus, the lower bound is included and the upper bound is excluded from the specified interval. A CTL input file may also contain an arbitrary number of single-line and multi-line comments in $C^{++}$ style, allowing for better readable requirement specifications.

Additionally, there are a couple of non-standard temporal operators (EY, EH, FwdUntil, FwdGlobal), which, however, are beyond the scope of this introductory overview; see [Tov08] for details. They specifically allow for efficient forward traversal model checking (in preparation); compare Section 4.3.

## 3 Basic Data Structures

The tool is built upon an object-oriented implementation of Zero-suppressed Binary Decision Diagrams (ZBDDs) and Interval Decision Diagrams (IDDs). Both data structures are crucial for the reached performance, see Section 6. Here we sketch the basic principles to illustrate the compression effect, which we hope to gain generally by symbolic representations of very large state spaces.

ZBDDs [Min93] trail the success story of Binary Decision Diagrams (BDDs) [Bry86]. They are a special variation dedicated to the efficient representation of vast sets of sparse arrays, thus they are perfectly suited for the analysis of 1-bounded Petri Nets. The efficiency gain in comparison to standard BDDs comes from a special reduction rule which eliminates all variables that do not occur in a given marking (place is empty), compare Figure 2.

IDDs are a rather straightforward generalization of BDDs [Rid97], [ST98]. Arcs are labelled by (possibly) real intervals, the number of outgoing arcs of a node can vary, and values of IDD variables are not bounded. To analyse k-bounded Petri Nets, Boolean IDDs (IDDs with only two terminal nodes: 0 and 1) over integer intervals are used, compare Figure 3. Every Boolean IDD over $n$ variables represents a function $f$ that can be written as an interval logic formula over $n$ variables, i.e., as a formula containing only atomic propositions over integer variables combined by logic operators. To find the result of a function for the given values $a_i$ of all variables $x_i$, one follows a path through the graph from the root to a terminal node. In a non-terminal node $v$, an edge labelled with $I_j$ must be chosen if $var(v) = x_m$ and $a_m \in I_j$ . The result of the function is defined by the label of the terminal node reached. As usual, all paths to the terminal node 1 can be read as the encoding of a state set.

ZBDDs and IDDs enjoy the same powerful property in that they yield a canonical representation of Boolean or interval logic functions, respectively, if they are ordered and reduced. Boolean functions naturally encode sets of states of 1-bounded Petri nets, while interval logic functions allow a natural encoding of sets of states of k-bounded Petri nets. Both types of decision diagrams allow

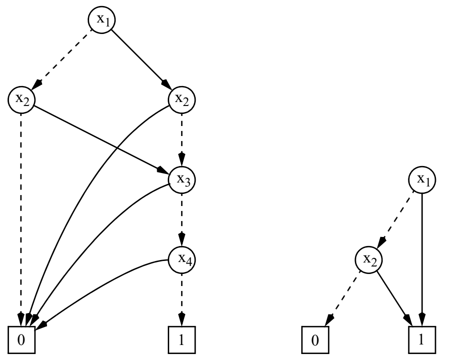


**Fig. 2.** The state set $\{ (1,0,0,0), (0,1,0,0) \}$ in BDD and ZBDD representation

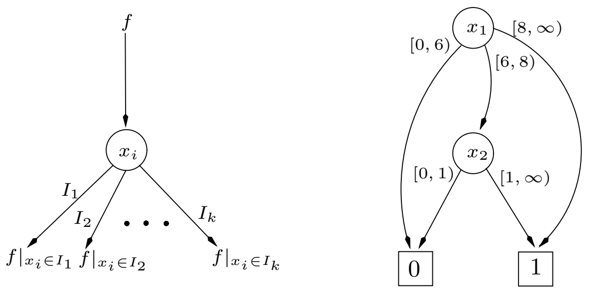


**Fig. 3.** Bool-Shannon decomposition for IDDs; general principle (left), and the IDD encoding the interval logic function $f = (x_1 \geq 8) \vee (x_1 \in [6,8) \wedge x_2 > 0)$ (right)

straightforward implementation of the basic operations required for the various Petri net analyses.

All algorithms and options sketched in the next section are equally available for both data structures and, thus, for 1-bounded and k-bounded Petri nets.

## 4 Main Features

There is a wide variety of tool options among which the user can choose. We sketch here only the most important ones. According to our experience up to now, there seems to be no general rule for the best choice of options.

### 4.1 Preliminaries

The **variable ordering** (i.e. place ordering) is known to have a strong influence on the decision diagrams' size and, thus, on the computation speed. A bad choice may even totally prevent the state space's constructability. There is no general rule for the best method and we provide several options:

1. plain order as read from the APNN file
2. reverse order to the one read from the APNN file
3. a random order
4. heuristic 1 – an algorithm computing weights for places based on the net structure [Noa99]
5. heuristic 2 – an adaption of the previous one [Tov08] (default)
6. derived from the transition ordering
7. read from a file as specified by the user

Likewise, the **transition ordering** has a crucial impact on the performance of the chaining and the saturation algorithms. There are several options: plain, random, read from file, and three heuristics derived from the net structure (one of the heuristics is the default).

There are three categories of **state space generation algorithms.**

1. Common Breadth-First Search (BFS): an iteration fires sequentially all transitions (according to the transition ordering) before adding the new states to the state space.
2. Transitions chaining: like BFS, but the state space is updated after the firing of each single transition.
3. Saturation algorithm (SAT): transitions are fired in conformance with the decision diagram, i.e. according to an ordering, which is defined by the variable ordering. A transition is saturated if its firing does not add new states to the current state space. Transitions are bottom-up saturated (i.e. starting at the terminal nodes and going towards the root). Having fired a given transition, all preceding transitions have to be saturated again, either after a single firing (single) or the exhausted firing (fixpoint) of the current transition; see Section 6 for some benchmarks.

### 4.2 Analysis of General Behavioural Properties

No a priori knowledge of the precise **boundedness** degree is required. The boundedness of each individual place as determined by the state space construction can be written to a file.

Symbolic algorithms for the computation and enumeration of the terminal **strongly connected components** allow one to determine efficiently liveness and reversibility. All live/nonlive transitions can be written to a file.

The **dead state analysis** with trace generation determines all reachable dead states, which can be written to file as an interval logic expression, i.e., as an expression containing only atomic propositions over integer variables combined by logic operators. All empty places do not appear in the expression. A transition sequence producing one of the reachable dead states can be written to a file as well.

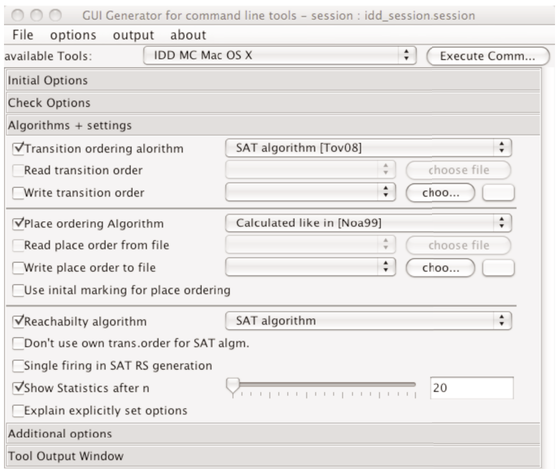
### 4.3 Model Checking of Special Behavioural Properties

**Limited backward reachability analysis**. The traditional backward analysis to check the operator $EU$ can often be improved by deleting all states after each backward step, which are not reachable from the initial marking. We support this established technique complemented by efficient **saturation-based reachability analysis**.

Furthermore, **forward traversal reachability analysis** is under preparation, which is expected to be more efficient in some cases than any backward analysis technique. It will be applied for all formulae containing the non-standard forward operators $FwdUntil$ and $FwdGlobal$, the past tense operators $EY$ and $EH$, and for all formulae which can be transformed using them; see [Tov08] for transformation rules.

## 5 Graphical User Interface

The symbolic analyser DSSZ-MC per se is implemented as a command line tool. However, it comes along with an optional Graphical User Interface (GUI) which is designed to assist the user in choosing among the various tool features [Fra08]. Actually, it is a general GUI generator for command line tools configured by an xml file specifying the individual tool options and their interdependencies, so it can be easily translated into other languages or adjusted to new tool options or even other tools. Figure 4 gives a snapshot of the sub-window offering the choice amongst the possible algorithms and related settings of DSSZ-MC.



available Tools: IDD MC Mac OS X

Initial Options

Check Options

Algorithms + settings

- [✓] Transition ordering alorithm: SAT algorithm [Tov08]
- [ ] Read transition order
- [ ] Write transition order
- [✓] Place ordering Algorithm: Calculated like in [Noa99]
- [ ] Read place order from file
- [ ] Write place order to file
- [ ] Use inital marking for place ordering
- [✓] Reachabilty algorithm: SAT algorithm
- [ ] Don't use own trans.order for SAT algm.
- [ ] Single firing in SAT RS generation
- [✓] Show Statistics after n: 20
- [ ] Explain explicitly set options

Additional options

Tool Output Window

**Fig. 4.** A snapshot of the GUI to choose among the various tool options

## 6 Benchmarks

We compare our tool with SMART [CS03] which is the best tool known for the symbolic analysis of k-bounded Petri nets with extended arcs. It deploys Multi-valued Decision Diagrams (MDDs) and also implements a saturation-based reachability algorithm. In contrast to DSSZ-MC, which handles monolithic Petri nets, SMART requires a suitable partitioning of the place set of the net to achieve good results. Unfortunately, defining a good partitioning is generally not a trivial task for non-regular models. SMART's saturation algorithm saturates MDD nodes, while DSSZ-MC applies the saturation strategy transition-wise to the whole decision diagram.

Our test suite comprises six Petri net models. The first three (philosophers, kanban, FMS) are taken from SMART's examples archive, which come along with a partitioning [MC99]. We added two biochemical networks [GH06], [HGD08], and a Petri net weakly computing the Ackermann function [PW03]. We tried our very best to find suitable partitionings for them.

**Table 1.** Some benchmarks for the ZBDD tool

| model | net | | | ZBDD-MC time secs | | SMART time secs |
|---|---|---|---|---|---|---|
| | \|P\| | \|T\| | \|states\| | fixpoint | single | |
| phils_N500 | 3000 | 2000 | 3.03e+313 | 12.77 | 12.83 | 7.60 |
| phils_N1000 | 6000 | 4000 | 9.18e+626 | 97.14 | 97.28 | 15.74 |

**Table 2.** Some benchmarks for the IDD tool [a)]

| model | net | | | IDD-MC time secs | | SMART time secs |
|---|---|---|---|---|---|---|
| | \|P\| | \|T\| | \|states\| | fixpoint | single | |
| kanban_N50 | 16 | 16 | 1.04e+16 | 14.68 | 0.54 | 386.00 |
| kanban_N75 | | | 7.83e+17 | - | 4.17 | 674.66 |
| kanban_N100 | | | 1.73e+19 | - | 6.50 | - |
| kanban_N200 | | | 3.17e+22 | - | 37.86 | - |
| kanban_N300 | | | 2.65e+24 | - | 265.01 | - |
| FMS_N100 | 22 | 21 | 2.70e+21 | 10.28 | 6.60 | 3.11 |
| FMS_N200 | | | 1.95e+25 | 38.09 | 57.41 | 37.56 |
| FMS_N250 | | | 3.46e+26 | 107.11 | 170.11 | 70.33 |
| FMS_N300 | | | 3.65e+28 | 907.37 | - | - |
| erk_N50 | 11 | 11 | 2.83e+8 | 2.74 | 2.27 | 25.23 |
| erk_N100 | | | 1.59e+10 | 3.99 | 2.51 | 231.41 |
| erk_N200 | | | 9.52e+11 | 29.80 | 4.08 | - |
| erk_N700 | | | 1.67e+15 | - | 55.18 | - |
| levchenko_N20 | 22 | 30 | 8.81e+10 | 2.34 | 2.39 | 6.82 |
| levchenko_N40 | | | 4.78e+14 | 3.00 | 2.44 | 133.23 |
| levchenko_N80 | | | 5.63e+18 | 17.44 | 3.34 | † |
| levchenko_N120 | | | 1.62e+21 | 153.95 | 5.73 | † |
| levchenko_N160 | | | 1.06e+23 | - | 10.88 | † |
| levchenko_N320 | | | 2.62e+27 | - | 133.54 | † |
| ack(3,2) | 23 | 24 | 1.44e+07 | 2.81 | 4.14 | 76.75 |
| ack(3,3) | | | 1.34e+09 | 6.79 | 32.97 | - |
| ack(3,4) | | | 1.42e+11 | 43.50 | - | - |

[a)] '–' means that physical memory was exhausted,
† we did not get results within one hour computation time.

The benchmarks were done on a 2 GHz Pentium M with 2 GB RAM running a 32bit Linux. In general, we used the default settings, which are in most cases the best possible choice. The memory option was set to 4 (up to 2.6 GB memory). It may be worthwhile to try different place ordering options to find the most suitable one. We did so for the kanban and FMS nets, for which we used option 6. Furthermore we tried the SAT algorithm with single firing first, which usually speeds up the construction significantly. Tables 1 and 2 show the total processing time for the state space computation using the saturation algorithms (fixpoint, single). These figures do not include the precious time a tool user spends to look for good options and suitable net partitionings.

The results suggest that the two tools under consideration may complement each other, depending on the power of the variable/transition orderings and net partitionings found.

## 7 Technicalities

The tool is a complete re-implementation in C$^{++}$, using the GNU MB Bignum Library (GMP). The parser of CTL formulae has been generated by the lexical analyser and parser generator *flex* and *bison*, respectively. The source code comprises about 22,800 LOC (Lines of Code, including comments) and has been tested on Windows, Linux, and MAC/OS.

The command line tool comes as two all-inclusive binaries (statically linked libraries), therefore, no special installation is required. Each takes about 1-2 MB memory, depending on the platform.

The GUI is written in Java and delivered by an installer as a Java jar file, so it requires the Java run-time environment (1.6 or higher).

The tool is available free of charge for scientific purposes at www-dssz.informatik.tu-cottbus.de. We provide the binaries for Windows, Linux, and Mac/OS. At the same website one also finds all the Petri net examples (in Snoopy, APNN, and SMART syntax) which we used as benchmarks in the preceding section. Maybe the reader finds better partitionings and/or options?

## 8 Conclusions

We have presented a tool for the symbolic analysis of extended Petri nets that supports the efficient analysis of general behavioural properties and CTL model checking as well. The models have to be bounded, however, no a priori knowledge of the precise boundedness degree is required. Crucial points for the tool's performance are the data structures used for the symbolic state space representation, and the algorithms, which exploit strongly connected components and the saturation principle.

We are working on a more detailed comparison with related tools, including liveness and reversibility decision as well as a representative set of model checking queries.

Besides the forward traversal strategy for efficient model checking mentioned in Subsection 4.3, we consider an extension of the current implementation by allowing a set of initial states. This set has then to be specified by an interval logic expression.

Continuing the encouraging results we have gotten so far, we are developing IDD-based model checking of Continuous time Stochastic Logic (CSL), see [Sch08]. A first prototype is available on the same website as the tool described in this paper.

## References

[BKK94] Bause, F., Kemper, P., Kritzinger, P.: Abstract Petri Net Notation. Technical report, Univ. Dortmund, CS Dep. (1994)

[Bry86] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Computers C-35(8), 677–691 (1986)

[CGP01] Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999) (third printing, 2001)

[CS03] Ciardo, G., Siminiceanu, R.: Structural symbolic CTL model checking of asynchronous systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 40–53. Springer, Heidelberg (2003)

[Fra08] Franzke, A.: A concept for redesigning Charlie. Technical report, BTU Cottbus, Dep. of CS (2008)

[GH06] Gilbert, D., Heiner, M.: From Petri nets to differential equations - an integrative approach for biochemical network analysis. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 181–200. Springer, Heidelberg (2006)

[HGD08] Heiner, M., Gilbert, D., Donaldson, R.: Petri nets in systems and synthetic biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 215–264. Springer, Heidelberg (2008)

[HRS08] Heiner, M., Richter, R., Schwarick, M.: Snoopy - a tool to design and animate/simulate graph-based formalisms. In: Proc. PNTAP 2008, associated to SIMUTools 2008. ACM digital library, New York (2008)

[KJH05] Koch, I., Junker, B.H., Heiner, M.: Application of Petri Net Theory for Modeling and Validation of the Sucrose Breakdown Pathway in the Potato Tuber. Bioinformatics 21(7), 1219–1226 (2005)

[MC99] Miner, A.S., Ciardo, G.: Efficient reachability set generation and storage using decision diagrams. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 6–25. Springer, Heidelberg (1999)

[Min93] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. 30th ACM/IEEE Design Automation Conference (DAC), pp. 272–277. ACM Press, New York (1993)

[Noa99] Noack, A.: A ZBDD package for efficient model checking of Petri nets. Technical report, BTU Cottbus, Dep. of CS (1999) (in German)

[PW03] Priese, L., Wimmel, H.: Theoretical Informatics - Petri Nets. Springer, Heidelberg (2003) (in German)

[Rid97] Ridder, H.: Analysis of Petri Net Models with Decision Diagrams. PhD thesis, University Koblenz-Landau (1997) (in German)

[Sch08] Schwarick, M.: Transient Analysis of Stochastic Petri Nets With Interval Decision Diagrams. In: Proc. 15th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2008), September 2008. CEUR Workshop Proceedings, vol. 380, pp. 43–48. CEUR-WS.org (2008)

[SSE03] Schröter, C., Schwoon, S., Esparza, J.: The Model Checking Kit. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 463–472. Springer, Heidelberg (2003)

[ST98] Strehl, K., Thiele, L.: Symbolic model checking using interval diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich (1998)

[Tov08] Tovchigrechko, A.: Model Checking Using Interval Decision Diagrams. PhD thesis, BTU Cottbus, Dep. of CS (2008)