GreatSPN Enhanced with Decision Diagram Data Structures

Junaid Babar[1], Marco Beccuti[2], Susanna Donatelli[2], and Andrew Miner[1]

[1] Department of Computer Science
Iowa State University
{junaid, asminer}@iastate.edu
[2] Dipartimento di Informatica
Universitá di Torino
{beccuti, susi}@di.unito.it

Abstract. Decision diagrams (DDs) have made their way into Petri net (PN) tools either in the form of new tools (usually designed from scratch to use DDs) or as enhancements to existing tools. This paper describes how an existing and established tool (GreatSPN) has been enhanced through the use of DDs provided by an existing open-source library (Meddly). We benchmark the enhanced tool and discuss lessons learned while integrating DDs into GreatSPN.

Category: tool paper

1 Introduction

Generalized Stochastic Petri Nets (GSPNs) [1] and Stochastic Well-formed Nets (SWNs) [4] are well-known extensions of Petri Nets (PNs). GSPNs are useful in modeling stochastic delays where transitions are either *immediate* or *timed*, i.e., they fire with a zero or an exponentially distributed delay. SWNs add token identities to GSPNs and the possibility of automatically exploiting symmetries for efficient state space generation.

GreatSPN is a suite of tools for the design and analysis (qualitative and quantitative) of GSPNs and SWNs. First released by the University of Torino in the late 1980's, GreatSPN has been a widely used tool in the research community, and remains so as it provides a breadth of solvers for computing net structural properties, the reachable states (RS), the reachability graph (RG) with and without symmetry exploitation, and performance evaluation indices using either simulation or analytical solution for steady-state and transient measures. While these solvers are efficient in enabling and firing operations and the underlying data structures are optimized, they do not take advantage of *symbolic* (implicit) storage techniques based on decision diagrams (DDs). A question then arises: can the use of symbolic storage techniques improve GreatSPN's performance while retaining *all* of its features? And if so, by how much and at what cost? Our final goal is to have a state-of-the-art tool that supports advanced data structures

for solving GSPN and SWN while saving memory and preserving or improving time. To limit costs and to ensure adequate implementation quality, we decided to use an existing DD library that automatically handles the complex aspects of using DDs such as caching and garbage collection. We selected Meddly [13], a new DD library, which provides a simple interface (in addition to an expert-level interface for low-level access). Our choice was motivated mainly by the variety of types of DDs that Meddly supports, and by certain features of the library (such as the ability to expand the set of possible values for a variable).

The contribution of this work is along two lines: to improve GreatSPN and to show that, with limited effort and care, an existing, structurally complex tool, can be enhanced with DDs through the use of a library. Being the first use of Meddly inside an existing tool, this work can also be seen as a template for integrating Meddly into existing tools.

There are a number of PN tools that use DDs as preferential data structures. To limit the scope of the related work we shall only consider PN tools and in particular (G)SPN tools since their solutions pose some additional challenges to DDs (as discussed later). The interested reader may refer to [14] for an overview of the different variations of DDs employed by different tools.

SMART [18] was the first SPN tool to use DDs for RS generation and, later, for CTMC storage. It uses an efficient technique (*saturation* [5]) for state-space generation, and can store CTMCs in a variety of compact representations (including matrix diagrams and Kronecker algebra [14]). SMART has a number of additional features, like a rich language for model definition (that extends beyond classical PNs), a simulator, and a model checker for non-stochastic temporal logics; however, it lacks a GUI for net definition.

IDD-CSL [10, 17] is a tool targeted towards SPNs for system biology. It supports a rich language for transition rate definition but does not seem to allow immediate transitions or inhibitor arcs. It supports model checking of the stochastic logic CSL, and the computation of steady-state or transient performance indices. It belongs to a suite of tools that provide a GUI for defining a PN, and a tool for model checking non-stochastic logic and for computing a variety of structural properties of PNs. IDD-CSL uses IDDs (interval DDs) augmented with state indices (Labeled IDD) needed for computing performance indices. [17] discusses the necessary changes made to IDD for CTMC solution.

The data structures and associated algorithms of tools like SMART and IDD-CSL are very efficient in time and space; however, the DDs are embedded in the tools, and are not available to the community through a public library. Also, considering that these tools were designed with DDs in mind, they are usually, and unsurprisingly, faster than the enhanced GreatSPN.

A relevant example of a tool that has been enhanced through the use of DDs is Moebius [15], a tool for a superclass of GSPNs called Stochastic Activity Networks. Moebius allows for easy integration of different solution methods (and formalisms). A DD-based state space and CTMC generator and solver have been added, but according to the manual, the DD solution saves space but has a very high time penalty.

LibDDD[7] is a publicly available SDD [9] library that we considered as an alternative to Meddly. SDD is thought to be an effective data structure for SWNs, and is well suited for cases in which there is insufficient knowledge of the variables (number and domain) that define the state space to be generated. We chose to use Meddly instead since SDDs are more powerful than necessary for GSPNs and as stated by libDDD's authors, this power comes at a price. Also, libDDD only provides a low-level interface while Meddly provides (in addition to a low-level interface) a simple non-expert interface (which is all we needed).

The rest of the paper is organized as follows: Sec. 2 recalls the analysis engines of GreatSPN. Sec. 3 gives a brief overview of DDs and Meddly. Sec. 4 describes how GreatSPN has been modified to use DDs using Meddly and discusses the memory and time performance. Finally, Sec. 5 concludes the work, summarizes the lessons learned, and describes our future plans.

2 Overview of GreatSPN

GreatSPN is a suite of tools for the design and analysis of GSPNs and SWNs. Its analysis modules support the qualitative and quantitative analysis of GSPNs and SWNs through operations like computation of structural properties, state space generation and analysis, and analytical computation of performance indices. The first modules we chose to optimize were the state space enumeration algorithms, since they are common to both state space analysis and computation of performance indices. GreatSPN uses different solvers for GSPN and SWN. We decided to start our enhancement work from the GSPN solvers for two main reasons: first, although they are theoretically simpler than the ones for SWN, they are a difficult test for the integration of Meddly into GreatSPN as the data structures are optimized and not always trivial to manipulate and understand; second, this code is more likely to suffer from software obsolescence, so that a rewriting is definitely beneficial.

GreatSPN follows a classical fixed-point algorithm for reachability graph (RG) generation, shown in Fig. 1, where $\mathcal{S}$ is the set of visited markings (the reachability set RS at the end), while $\mathcal{U}$ is the set of unexplored markings. For each marking m' added to $\mathcal{U}$, the list of enabled transitions $\mathcal{T}_{m'}$ is stored along with m'. The list $\mathcal{T}_m$ is utilized while constructing $\mathcal{T}_{m'}$, an important optimization for nets with a large number of transitions. Note that line 12 can be removed if only the reachability set, and not the reachability graph, is desired. The algorithm used by GreatSPN is actually more involved: if the Petri net contains *immediate* transitions, the *vanishing* markings are eliminated during generation, producing the set of *tangible* reachable markings; these details are omitted to simplify the presentation. GreatSPN keeps on a separate file a list of all tangible markings reachable from a given vanishing marking, so it is unnecessary to recompute them when re-entering a vanishing marking.

Major data structures for the algorithm include $\mathcal{S}$ and $\mathcal{U}$, while crucial operations include addition to $\mathcal{S}$, addition and removal of markings for $\mathcal{U}$, the test to decide if a marking already belongs to $\mathcal{S}$, and the computation of enabled

```
GenerateRG(marking m_0, Petri net PN)
 1: S ← {m_0};                                   • m_0 is the initial marking
 2: U ← {m_0};
 3: while U ≠ ∅ do
 4:    Remove some m from U;
 5:    Determine set T_m of enabled transitions in marking m;
 6:    for all t ∈ T_m do
 7:       Determine marking m' reached from m when t fires;
 8:       if m' ∉ S then
 9:          S ← S ∪ {m'};
10:          U ← U ∪ {m'};
11:       end if
12:       Add edge (m, m', t) to RG;
13:    end for
14: end while
15: return S, RG;
```

Fig. 1. Traditional enumeration algorithm to build the reachability graph

transitions. GreatSPN uses a balanced binary tree (BBT) for $\mathcal{S}$, and to further reduce memory, $\mathcal{S}$ only contains indices to position in a file (called .mark) that stores markings. To save disk space and I/O time, the markings are stored in a compact way, using an encoding algorithm that exploits P-invariants. Each entry in the BBT also contains the list of the transitions enabled in that marking. The unexplored markings in $\mathcal{U}$ are stored as a list built on the nodes of the BBT, so that for each marking in $\mathcal{U}$ the list of enabled transitions is readily available. Thus, lines 4 and 5 in Fig. 1 can be executed very quickly, but line 10 requires determining the enabled transitions in marking m', as explained above.

GreatSPN builds the *tangible* reachability graph (TRG), i.e., the graph after elimination of vanishing markings. During generation, the TRG edges (cf. line 12 in Fig. 1) are written to a file (called .wnrg), together with the associated rate (computed from the timed transition rate, possibly multiplied by the weight of the immediate path followed). The TRG file (.wnrg) and the compacted marking file (.mark) are the only information (together with the net and performance indices definitions) needed by the numerical solution engines.

To simplify these and other engines, each reachable marking is assigned a unique integer index in the set $\{0, 1, \ldots, |\mathcal{S}| - 1\}$. GreatSPN uses "discovery order" (the order in which markings are inserted into $\mathcal{S}$) to index the markings. Indices are required for CTMC construction and solution, and a mapping between each marking and its index is required in many contexts of the solution process, a typical example being the need to compute performance indices from the steady state solution vector of the CTMC. GreatSPN never stores explicitly the mapping between a marking and its index: the i^{th} marking is in the i^{th} position inside the .mark file, and the edges from the i^{th} marking are listed in the i^{th} record of the .wnrg file. Note that indices are *not* required for (non-stochastic) model checkers: as we shall see, keeping indices may have a significant price.

3 Overview of Meddly

Decision diagrams are directed acyclic graphs used to represent functions on a finite number K of variables, where each variable x_k can assume a finite number n_k of values. Nodes are either *terminal* nodes, which have no outgoing edges, or are *non-terminal* nodes, which are labeled with a variable. Different variable types and ranges, and different rules for managing the graphs lead to various forms of DDs. For instance, binary decision diagrams [3] represent boolean functions on boolean variables, of the form $f : \{0,1\}^K \to \{0,1\}$.

Meddly, short for Multi-way and Edge-valued Decision Diagram Library, is an open-source software library [13] that supports several types of DDs (varieties relevant to this work are discussed below). All forms of DDs in Meddly eliminate *duplicate* nodes (two nodes that represent the same function) and require *ordering* of nodes (there is a total ordering $\succ$ on the function variables). Furthermore, the set of possible values for variable x_k is assumed to be $\mathcal{D}_k = \{0, 1, \ldots, n_k - 1\}$. In Meddly, an ordered collection of variables with specified sizes is called a *domain*, which we write as $\mathcal{D} = \mathcal{D}_K \times \cdots \times \mathcal{D}_1$.

A named collection of nodes of a particular variety of DD, and associated with a common domain, is called a *forest*. Within a given forest, Meddly automatically eliminates duplicate nodes using a unique table [2], imposes other forest-specific reduction rules, and handles memory management of the nodes (storing them compactly, garbage collection, etc.). The following types of forests are relevant to this work and are supported in Meddly.

MDD: multi-way DD [11], for functions of the form $f : \mathcal{D} \to \{0,1\}$.

MTMDD: multi-terminal MDD for functions of the form $f : \mathcal{D} \to \mathcal{R}$ with either $\mathcal{R} \subset \mathbb{N}$ or $\mathcal{R} \subset \mathbb{R}$. The function "return values" are stored in the forest within the terminal nodes [8].

EV+MDD: edge-valued MDD for functions of the form $f : \mathcal{D} \to \mathbb{N} \cup \{\infty\}$. The function "return values" are stored in the forest along the edges and are summed together along paths in the graph [12].

Other forms are already supported or are planned for future releases.

An important feature of any DD software is the ability to create new functions through various operations. In Meddly, several operators are supported whose arguments are functions with a common domain. Additionally, Meddly provides operators to create, evaluate, and destroy functions in a forest. The operations relevant to this work are described below, using simplified versions of the functions (rather than the exact function prototypes) to clarify the presentation.

- `createEdge()` builds a function by explicitly stating a return value for a set of variable assignments. Multiple variable assignments and return values may be specified. Any unspecified assignments are assumed to return a default value (normally 0 but dependent on the forest type). Thus, within a forest F, a call to F`.createEdge(`$(a_K, \ldots, a_1)$`,` a`,` $(b_K, \ldots, b_1)$`,` b`)` produces a

representation of function

$$f(x_K, \ldots, x_1) = \begin{cases} a & \text{if } x_K = a_K \wedge \cdots \wedge x_1 = a_1 \\ b & \text{if } x_K = b_K \wedge \cdots \wedge x_1 = b_1 \\ 0 & \text{otherwise} \end{cases}$$

within forest F.

- `apply()` builds a function by applying some operator on a set of operands. In particular, for an element-wise binary operator $\oplus$, the function

$$f(x_K, \ldots, x_1) = g(x_K, \ldots, x_1) \oplus h(x_K, \ldots, x_1)$$

can be obtained by calling `apply(`g`, `$\oplus$`, `h`, `f`)`. This assumes that functions f, g, and h have the same domain (they can be in different forests) and that the operator $\oplus$ is defined for the range of g and h. Similarly, for an element-wise unary operator $\ominus$, the function

$$f(x_K, \ldots, x_1) = \ominus g(x_K, \ldots, x_1)$$

can be obtained by calling `apply(`$\ominus$`, `g`, `f`)`. Meddly also provides symbolic state space generation algorithms (including a traditional iteration [16] and saturation [5]) by calling `apply` with an appropriate operator.

- `evaluate()` determines, for the representation of function f, the value of f for a given set of variable assignments. Specifically, `evaluate(`f`, `$(v_K, \ldots, v_1)$`)` gives the value of $f(v_K, \ldots, v_1)$.

Meddly automatically uses and maintains a *computed table* to reduce (often significantly) the computational cost of the `apply()` operations [2]. Meddly also allows for variables sizes to be increased as needed; this allows the user to start building a DD without knowing the *final* variables sizes. Finally, Meddly provides an expert-level interface so that users can define their own operations or access advanced features of the library.

4 Enhancing GreatSPN

We have experimented with a few different methods, described below, that we have developed (in an incremental manner) to evaluate the effectiveness of the DDs at various stages. For each method, the goal is to replace the existing representation of $\mathcal{S}$, namely the BBT and/or the .mark file, with DDs.

4.1 Changing only the RS

As a first step, we utilize the existing explicit state space generation algorithm in GreatSPN, rather than discarding it in favor of an entirely "symbolic" algorithm (such as [5, 16]). We assign an ordering to the set of places $\mathcal{P}$ (based on the order in which places are defined in the input file), and build the function

$$f(x_{|\mathcal{P}|}, \ldots, x_1) = 1 \quad \text{iff} \quad \exists m \in \mathcal{S} : m(p_{|\mathcal{P}|}) = x_{|\mathcal{P}|} \wedge \cdots \wedge m(p_1) = x_1$$

N	$\lvert\mathcal{S}\rvert$	Original			MDD for $\mathcal{S}$		MDD & $\mathcal{N}$		MTMDD		EV+MDD	
		BBT	File	T.	Mem.	T.	Mem.	T.	Mem.	T.	Mem.	T.
Dining philosophers Petri net (PHIL)												
7	2.4×10^4	1,175	1,027	8s	51	8s	36	0.13s	3,449	9s	74	18s
8	1.0×10^5	4,977	4,976	45s	75	50s	49	0.15s	14,607	38s	111	87s
9	4.3×10^5	21,082	23,717	345s	103	411s	64	0.18s	61,872	210s	160	421s
10	1.8×10^6	89,304	104,654	31m	139	28m	80	0.24s	262,092	18m	222	34m
11	7.8×10^6	—	—	—	185	82m	99	0.35s	1,084,212	78m	299	156m
12	3.3×10^7	—	—	—	235	7h	120	0.43s	—	—	392	8h
30	6.4×10^{18}	—	—	—	—	—	829	10s	—	—	—	—
40	1.1×10^{25}	—	—	—	—	—	1,576	32s	—	—	—	—
50	2.2×10^{31}	—	—	—	—	—	2,364	1m	—	—	—	—
Flexible manufacturing system Petri net (FMS)												
4	$1,3 \times 10^5$	6,627	3,037	11s	76	14s	363	6s	14,740	11s	2,397	22s
5	6.5×10^5	31,466	14,421	134s	135	63s	775	27s	66,758	81s	9,744	212s
6	2.5×10^6	120,940	55,430	7m	218	3m	1,470	1m	246,234	5m	33,068	42m
7	8.2×10^6	—	—	—	353	12m	2,536	5m	625,221	19m	97,389	6h
8	2.3×10^7	—	—	—	515	32m	4,070	13m	—	—	—	—
9	6.1×10^7	—	—	—	—	—	6,179	31m	—	—	—	—
10	1.4×10^8	—	—	—	—	—	8,997	1h	—	—	—	—
11	3.3×10^8	—	—	—	—	—	12,688	2h	—	—	—	—

Table 1. Time and memory (Kb) required for generation

as an MDD. The MDD representation for $\mathcal{S}$ is built exactly as described in the algorithm of Fig. 1: if `evaluate(`f`,` m'`)` is equal to 0, then $m' \notin \mathcal{S}$; the operation $\mathcal{S} \leftarrow \mathcal{S} \cup \{m'\}$ can be performed by building a function g with a call to `createEdge(`m'`, 1)`, where g represents the set $\{m'\}$, and then calling `apply(`f`, +,` g`,` f`)` to union the sets.

To simplify changes in the implementation, we have kept the list structure for $\mathcal{U}$, which is now a list of pointers to the position in the .mark file. No indices are stored for the markings and therefore it is not possible to generate the reachability graph and the CTMC. However, this method can be used for reachability analysis, and more importantly, it allows us to check the efficiency of the MDD representation with minimal changes to the implementation.

Results on two benchmark Petri nets are reported in Table 1. In the N dining philosophers (taken from [16]), the numbers of places, transitions, and MDD variables increase linearly with N, while in the flexible manufacturing system [6], the model parameter N specifies the number of parts (initial tokens in certain places), and the number of possible values for the MDD variables increases linearly with N. Experiments were run on an 2.4 GHz AMD Athlon 64-bit processor with 4 GB memory capacity. In the table the "Original" columns refer to the original GreatSPN implementation, and show the memory required in Kilobytes for BBT and for the .mark file, while "MDD for $\mathcal{S}$" refers to this first enhancement (use of an MDD for $\mathcal{S}$ and a list for $\mathcal{U}$). For the methods based on DDs, the reported memory is the "peak" memory use; the "final" memory

use can be less (often substantially so). From the table it is clear that a first objective has been achieved, since memory consumption is significantly reduced, while time is a bit better than with plain GreatSPN, but still it does not allow us to obtain the huge state spaces that DDs can often achieve. For instance, we cannot solve FMS with $N = 9$ due to the huge size of the .mark file.

A clear cause for the time bottleneck is that states are added to $\mathcal{S}$ one at a time, so that execution time is at least linear in the number of states. To overcome this limit, a "symbolic" firing has been implemented, based on a MDD representation of the next-state function $\mathcal{N}$, that allows for the efficient determination of all states reachable from the states in the current $\mathcal{S}$ set in a single firing. In particular, this requires to encode $\mathcal{N}$ with an MDD that has twice as many variables as the MDD that encodes the RS, since it represents transitions between states. In our implementation we use the inhibition, pre- and post- incidence matrices (available in GreatSPN) to derive $\mathcal{N}$ for the model. The RS is then generated by Meddly by calling `apply(`*rsgen*`,` m_0`,` $\mathcal{N}$`,` $\mathcal{S}$`)`, which invokes the traditional symbolic algorithm [16] on initial marking m_0 with next-state function $\mathcal{N}$, and stores the result in $\mathcal{S}$.

Results are reported in columns "MDD & $\mathcal{N}$ " of Table 1. Note the large saving in time and space for the PHIL model, while for FMS the results are less impressive, which in a way is not surprising: FMS includes priorities for transitions, leading to a more involved implementation of the symbolic firing, which also requires the use of some additional, intermediate, MDD. These intermediate MDDs are included in the peak memory usage reported in the table.

4.2 MTMDD or EV+MDD for reachability graph

If we wish to build the reachability graph, and the CTMC, we need a new mechanism for remembering the index of each discovered marking in $\mathcal{S}$. This can be done by constructing the function (with slight abuse of notation)

$$f(m) = \begin{cases} 0 & \text{if } m \notin \mathcal{S} \\ \text{index of } m & \text{if } m \in \mathcal{S} \end{cases}$$

where the marking indexes are distinct integers in the range 1 to $|\mathcal{S}|$, assigned in "discovery order". This function can be represented either as a MTMDD or an EV+MDD. In either case, the implementation follows the discussion in Sec. 4.1, except when adding marking m' to $\mathcal{S}$, we call `createEdge(`m'`,` *index*`)`. Note that this means that we are back to generating one state at a time.

For both solutions (MTMDD and EV+MDD) we have also modified the implementation of $\mathcal{U}$ to use an MDD. The operation "remove some m from $\mathcal{U}$" is implemented through a Meddly function `findFirstElement()`, that efficiently finds a function's "first" set of variable assignments that return a non-zero value; this marking is then removed from $\mathcal{U}$ via `apply()` with the subtraction operator.

As a consequence the list of enabled transitions is not stored any longer (to do so would require encoding a different function f with extra variables for the set of enabled transitions). We therefore must determine the enabled transitions

by checking them all when exploring a marking, rather than by simply updating the known enabled transitions when a new marking is entered (in the algorithm of Fig. 1 line 5 becomes more expensive, while line 10 becomes less expensive).

The performance of the implementation of RS with indices, using MTMDD and EV+MDD, is reported in the last columns of Table 1. The MTMDD turns out to be a worst-case scenario: since each marking has a unique index, there will be no sharing of nodes in the MTMDD. The EV+MDD, instead, can be shown to be never worse than the MTMDD, and for many functions, can be exponentially better. However, manipulating EV+MDDs has a higher cost than MTMDDs, since additional operations are required on the edge values. This classical time-memory tradeoff is clearly seen in Table 1: the MTMDD implementation always requires less time and more memory than the EV+MDD implementation. Interestingly, the MTMDD implementation usually outperforms the original one, even without the faster mechanism to determine the enabled transitions.

5 Conclusion

In this paper, we described how an existing and established tool, GreatSPN, was enhanced through the use of DDs as provided by an existing open-source library (Meddly). Several important lessons can be taken away from this exercise. The enhancement described has been implemented using only the non-expert interface of Meddly, which implies a limited learning curve, an implementation that is easier and therefore more likely to be error-free, and the possibility to quickly exploit any future development of Meddly. To provide a feeling of how easy it is to experiment with Meddly, the implementation for MTMDD and for EV+MDD differ only in a single parameter in the forest creation.

Our first attempt using MDDs tells us that improvements can be achieved with minimal changes, but that better improvements can be achieved if the generation algorithm is changed to exploit DD strength (e.g., a symbolic algorithm using $\mathcal{N}$), since any savings gained through the optimal use of DDs may be enough to offset the loss of any optimizations that must be discarded.

We have several plans for further enhancement of GreatSPN with DDs. The issue of index storage for RG and CTMC generation and solution requires further investigation. A possible modification is to store the reachability graph directly in a DD; this would allow the use of MDDs (instead of MTMDDs or EV+MDDs) since indexes would not be required until after generation. However, this would require to either convert the DD representation of the CTMC back into the data structure currently used by GreatSPN, or to completely re-implement all the numerical solution engines to use DDs. Either of these requires advancements in Meddly and substantial implementation in GreatSPN. A different line of work is the extension from GSPN to SWN, this however will require the ability to deal with markings that do not have a fixed, bounded structure, for which SDDs may be a better choice than MDDs. As such, we plan to investigate the use of libDDD[9] for state space generation of SWNs and compare with Meddly.

Acknowledgment

This work is supported in part by the National Science Foundation under grant CNS-0546041.

References

1. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. J. Wiley (1995)
2. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: 27th ACM/IEEE Design Automation Conference. pp. 40–45. ACM Press (1990)
3. Bryant, R.E.: Graph–based algorithms for boolean function manipulation. IEEE Trans. Comput. C-35(8), 677–691 (Aug 1986)
4. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: Stochastic well-formed coloured nets for symmetric modelling applications. IEEE Trans. Comput. 42(11), 1343–1360 (Nov 1993)
5. Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state–space generation. Formal Methods in System Design 31(1), 63–100 (Aug 2007)
6. Ciardo, G., Trivedi, K.S.: A decomposition approach for stochastic reward net models. Perf. Eval. 18, 37–59 (1993)
7. LibDDD webpage. `http://move.lip6.fr/software/DDD`
8. Fujita, M., McGeer, P., Yang, J.Y.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. Formal Methods in System Design 10(2–3), 149–169 (Apr 1997)
9. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical set decision diagrams and automatic saturation. In: 29th Int. Conf. on Applications and Theory of Petri Nets. pp. 211–230. Springer-Verlag (Jun 2008)
10. IDD-CSL webpage. `http://www-dssz.informatik.tu-cottbus.de`
11. Kam, T., Villa, T., Brayton, R., Sangiovanni-Vincentelli, A.: Multi–valued decision diagrams: theory and applications. Multiple-Valued Logic 4(1–2), 9–62 (1998)
12. Lai, Y.T., Pedram, M., Vrudhula, S.: Formal verification using edge-valued binary decision diagrams. IEEE Trans. Comput. 45(2), 247–255 (Feb 1996)
13. MEDDLY webpage. `http://sourceforge.net/projects/meddly`
14. Miner, A., Parker, D.: Symbolic representations and analysis of large state spaces. In: Validation of Stochastic Systems, pp. 296–338. LNCS 2925, Springer-Verlag (2004)
15. Moebius webpage. `http://www.mobius.illinois.edu`
16. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri Net Analysis Using Boolean Manipulation. In: 15th Int. Conf. on Application and Theory of Petri Nets. pp. 416–435. Springer-Verlag (1994)
17. Schwarick, M., Heiner, M.: CSL model checking of biochemical networks with interval decision diagrams. In: 7th Int. Conf. on Computational Methods in Systems Biology. pp. 296–312. Springer-Verlag, Berlin, Heidelberg (2009)
18. SMART webpage. `http://www.cs.ucr.edu/~ciardo/SMART`