StrataGEM: A Generic Petri Net Verification Framework

Edmundo López Bóbeda, Maximilien Colange, and Didier Buchs

Centre Universitaire d'Informatique
Université de Genève
7 route de Drize, 1227 Carouge, Suisse

Abstract. In this paper we present the Strategy Generic Extensible Modelchecker (StrataGEM), a tool aimed at the analysis of Petri nets and other models of concurrency by means of symbolic model-checking techniques. StrataGEM marries the well know concepts of Term Rewriting (TR) to the efficiency of Decision Diagrams (DDs). TR systems are a great way to describe the semantics of a system, being readable and compact, but their direct implementation tends to be rather slow on large sets of terms. On the other hand, DDs have demonstrated their efficiency for model-checking, but translating a system semantics into efficient DDs operations is an expert's matter. StrataGEM describes the semantics of a system in terms of *strategies* over a TR system, and automatically translates these rules into operations on DD to handle the model-checking. The ultimate goal of StrataGEM is to become a verification framework for the different variants of Petri nets by separating the semantics of the model from the computation that performs model-checking.

1 Introduction

Decision Diagrams (DDs) have demonstrated their efficiency towards model-checking [1], especially for Globally Asynchronous Locally Synchronous (GALS) systems. However, they require a careful encoding of the operations at the DDs level. Most works in the area focus on a low-level encoding, which is a major obstacle for non-expert modelers.

High-level interfaces, easily understandable for the user not familiar with DDs, are desirable in order to ease and widen the use of DDs. With model-checking as goal, such an interface should provide a language for the description of the semantics of the systems to be evaluated, and an automatic translation of this semantics into DDs operations. It should also feature some predefined model-checking algorithms, along with opportunity to define new algorithms, *e.g.,* through the language used for the semantic description. This language should meet two criteria: be expressive enough to capture the largest possible class of systems, and be easy to read and write for a human being. This aim is similar to the principles used in successful SAT based model-checker such as bounded model-checker where there is a separation between the encoding of the problem in propositional logic and the SAT engines themselves.

G. Ciardo and E. Kindler (Eds.): PETRI NETS 2014, LNCS 8489, pp. 364–373, 2014.

StrataGEM is a prototype of such an interface, relying on Term Rewrite Systems (TRSs) as its language for semantic description, to achieve both expressiveness and readability. Term Rewriting (TR) rules are especially well-suited to describe local modifications, which suits asynchronous systems. Standard TR rules however suffer two drawbacks. First, GALS systems often feature various degrees of synchronization among local modifications; such synchronizations are not easy to express with standard TR rules. Second, model-checking algorithms often tweak the semantics of the system to increase performance, *e.g.,* by prioritizing concurrent transitions. Such operations are not easy to describe with standard TR rules either.

StrataGEM adresses these problems thanks to *TR strategies*. They enrich the language by allowing to combine standard TR rules in various ways. Thus, the semantics of a system is given in two steps: standard TR rules describe atomic semantic steps, and strategies using these atoms describe more elaborate transitions (such as synchronizations), or semantic tweaks for optimization [2,3].

Strategies also allow to describe model-checking algorithms, either built-in in the tool or designed by the user. Thus, semantics of systems and model-checking algorithms are treated uniformly, considerably easing the use of the tool.

As the use of TRS to describe the semantics of a system assumes to represent states of said system as terms, the natural choice for the underlying DDs is Σ Decision Diagrams (ΣDDs) [4]. Designed to represent efficiently large sets of terms, they come with basic efficient manipulation operations. In particular, they allow to apply a TR rule to a set of terms in one elementary step.

Thanks to its extensibility, StrataGEM is not only aimed at people wanting to model-check Petri nets, but also at developers of model-checkers, willing to implement their algorithms on top of ΣDD.

This paper is structured as follows: in Section 2 we informally present transition systems, based on TRS and strategies, as defined in StrataGEM. Section 3 presents the tool architecture and also introduces its basic usage. Section 4 presents an assesment of the tool, comparing it to another similar model-checking tool. The results are quite promising, the comparison with PNXDD indicating a trend towards a better asymptotic performance for StrataGEM.

2 StrataGEM Transition Systems

In this section, we informally present how to describe Transition System (TS) in StrataGEM. We first introduce the TRS and rewrite rules, that serve as basic bricks to describe local evolutions of the system. We then present the strategies that allow to combine these building blocks into elaborated semantics and algorithms. Their purpose is threefold: they describe the non-local transformations of the system, the model-checking algorithms, and the optimizations that can be done in these algorithms. Terms, that describe the states of the system, rewrite rules and strategies, that describe its semantics, are gathered to form a StrataGEM Transition System. To support our presentation, we use as running example the Kanban model, expressed as a Petri Net on Figure 1.

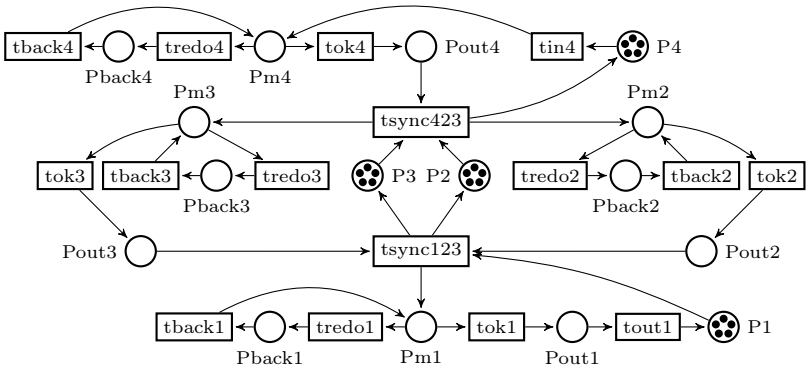

Fig. 1. Kanban Petri Net

2.1 Term Rewrite Systems

The first step is to define a signature: a set of function names used to build terms that encode the states of the system. Several encodings with different signatures may be possible. The construction of terms in StrataGEM follows the theory of Order-Sorted Signatures [5], which we do not detail here.

Let us describe the signature used by StrataGEM for Petri nets. Non-negative integers are encoded as terms being either `zero`, or `suc(t)` where `t` is an integer term. For clarity, in the following integer terms are often replaced by plain integers.

For a given Petri net, a function name of arity 2 is defined for each of its places. The first argument is the marking of the place (an integer term), and the second argument is the subsequent list of places. The function name `empty` of arity 0 denotes the empty list. The state (marking) of a Petri net is then a term containing all places exactly once with their respective number of tokens. Thus, in our running example, an initial state would start with `Pm4(0, P4(5, Pback (0, ...)))`, indicating that places `Pm4` and `Pback` initially contain no token, whereas place `P4` contains 5 tokens.

Basic local operations of the system are described in terms of rewrite rules, and aggregated with strategies. A rewrite rule `l` $\rightsquigarrow$ `r` states that if a term `t` matches its left-hand side `l`, then it is replaced by its right-hand side `r`. Note that both `l` and `r` may contain term variables. In the case of Petri nets, each arc gives rise to a rewrite rule, a transition of the net is then described by a strategy aggregating its adjacent arcs. For example, the arc removing a token from `Pm4` (either to fire transition `tredo4` or `tok4`) is described by the rewrite rule `arc1` = `Pm4(suc($x), $p)` $\rightsquigarrow$ `Pm4($x, $p)`, where values preceded by a dollar sign denote variables. Similarly, an arc adding a token in the place `Pm4` (either from transition `tback4` or `tin4`) is described by `arc2` = `Pm($x, $p)` $\rightsquigarrow$ `Pm4(suc($x), $p)`. Note that StrataGEM only handles linear rewriting rules, *i.e.,* that do not contain multiple occurrences of a variable in its left and right sides.

2.2 Term Rewriting Strategies

Expressing the Petri Nets transitions in terms of rewrite rules would be cumbersome, especially if the places modified by the transition are not close in the state vector. To cope with this problem we use Term Rewriting strategies [6,7], that are meant to control the application of rewrite rules. A TR rule might succeed and modify the term on which it is applied or fail, in which case the strategy returns nothing. A fail in a strategy can be seen as an exception that is raised.

Basic Strategies. The *simple strategy* wraps a standard rewrite rule as a strategy, and applies it at the *root* of a given term. Thus, a simple strategy is defined for each arc of the Petri Net, that embeds the arc rewrite rule described above.

Note that a simple strategy only applies its embedded rewrite rule at the top of a term. If the rule is applicable to a subterm, but not to the root, then the simple strategy fails. We denote the application of a strategy with square brackets. For example, `arc1[Pm4(suc(0), empty)]` = `Pm4(0, empty)`. However, `arc1[P4(suc(0), Pm4(suc(0), empty))]` will fail, because the term where it can be applied is a *subterm* of the root term, and not the root term itself.

Such basic strategies are sufficient to express the semantics of Petri Nets arcs, and we present in the following how to combine them to express the semantics of a whole transition. StrataGEM allows a more generic setting for simple strategies. A simple strategy embeds an ordered list of rewrite rules, and applies only the first of the rules that is applicable. We do not use this feature in the paper, but it is quite useful when several rewrite rules are applicable to the same term. The ordered list describes a priority between rules to resolve any ambiguity. If no rule in the list is applicable at the root, then the simple strategy fails.

Simple strategies serve as atomic building blocks for the transition relation. To combine them, StrataGEM provides a set of basic strategies, mostly inspired from Tom strategy language [7]. The user may also define custom strategies.

We introduce three basic strategies, and then combine them to apply an arc rewrite rule at the appropriate, but not *a priori* known, depth. The first of these basic strategies, `One(S,n)`, applies the strategy `S` to the n-th subterm of the root. This overcomes the limitation of simple strategies to apply only at the root of a term. The second strategy is the *choice strategy*, a simple conditional application of strategies: `Choice(S1,S2)[t]` tries to apply `S1` to `t`. It returns `S1[t]` if this application succeeds, and `S2[t]` if `S1` fails.

We now the third define a strategy *recursively*. It is for instance useful for the recursive propagation of a strategy on subterms, to apply it at an arbitrary depth. For instance, to apply an arc rewrite rule at the appropriate depth, we use the recursively defined strategy `ApplyOnce(S) = Choice(S, One(ApplyOnce(S),2)`. `ApplyOnce(S)` thus tries to apply `S` at root level. If it fails, it tries to apply it on the second subterm. If this also fails, given the definition of our state terms, it means that it was applied on `empty` and legitimately fails. This strategy thus descends the subterm tree until it finds a subterm where it can be applied. Note that this does not require to know the depth where the strategy `S` should be applied: this application depth is discovered dynamically.

We have described how to apply an arc rewrite rule at the appropriate depth. We now describe how to encode the semantics of a Petri net transition. It is simply the synchronization of the rewrite rules for all the incident arcs. This synchronization is performed with the *sequence strategy*: `Sequence(S1,S2)` first applies `S1`, then `S2`. If one of its arguments fails when applied, the sequence strategy also fails. To improve readability, it is also possible to define a n-ary sequence strategy, that applies its arguments one after the other. Thus, if a transition t of a Petri net has `iarc1`, ..., `iarcN` as input arcs and `oarc1`, ..., `oarcP`, then its semantics is given by the strategy `Sequence(iarc1, ..., iarcN, oarc1, ..., oarcP)`. This ensures that the input arcs rules are applied before the output arcs rules, thus ensuring the enabling precondition: if one input place does not contain enough tokens, the whole strategy fails.

Note that this encoding could be further improved by releasing constraints on the order the arc rewrite rules are evaluated, so as to avoid unnecessary walks along the term tree structure. The only constraint that should be retained is that the enabling precondition be retained. Another improvement would be the design of a strategy for a commutative sequence of strategies, that would determine dynamically the order in which the arc rewrite rules are applied.

Model-Checking Strategies. Strategies are not only meant to describe the transition relation of a system, but they can also describe how to explore the state space to check a property. We first extend the strategies to sets of terms: `S[T] = { S[t] | t ∈ T , S` does not fail on `t }`. The elements on which the strategy fails are removed from the resulting set. If the strategy fails on all terms in the set, then it fails on the set.

We define two more strategies to better handle sets of terms. The first one allows to gather the results of several strategies: `Union(S1,S2)[T] = S1[T] ∪ S2[T]`. The union fails if either `S1` or `S2` fails. As previously done for the sequence strategy, we use an n-ary union to ease readability. The order of its arguments is not relevant, since it is commutative.

`Fixpoint` applies a strategy to a set of terms until a fixpoint is reached: `Fixpoint(S)[T]` $= \mathtt{S}^n\mathtt{[T]}$ where n is the smallest integer s.t. $\mathtt{S}^n\mathtt{[T]} = \mathtt{S}^{n+1}\mathtt{[T]}$. We use these strategies to encode the state space generation of a concurrent system, such as the Petri Net of Figure 1. Say that the concurrent system has `N` transitions, encoded as strategies `T1`, ..., `TN`. The strategy that generates the state space of such a system is: `Fixpoint(Union(Identity, Choice(T1, Identity), ..., Choice(TN, Identity)))`. The `Identity` strategy rewrites a term to itself. Its presence in the union is necessary to keep states computed so far. `Choice` keeps strategies from failing, which would make the `Union` fail.

Optimizing Strategies. Most model-checking algorithms perform challenging computations, and it is in our best interest to perform them optimally. When using DDs, there are three main ways to improve the strategies:

– performing less operations;

– exploiting locality;
– exploiting subparts of the system that share similar behavior.

Consider for instance the operation `ApplyOnce(arc1)` defined above. We recall that `arc1` = `Pm4(suc($x), $p)` ⇝ `Pm4($x, $p)`. If `Pm4` does not contain enough tokens, `arc1` fails, and `ApplyOnce(arc1)` continues its descent of the subterm tree, eventually stopping at the end of the tree. But, by construction of our state terms, if `arc1` fails at this very level, it will not apply at any other level. Hence the recursive descent of `ApplyOnce(arc1)` becomes useless. To avoid this unnecessary and costly descent, we can first check if the appropriate depth is reached, and apply `arc1` only if true. In this case, the failure of `arc1` is not pursued by a descent on the subsequent places. We introduce a conditional strategy: `ITE(S1, S2, S3)` [1] first applies `S1`, then it applies `S2` if successful. If the application of `S1` fails in the first place, it applies `S3` to the input term. The trick here is to use a strategy that checks whether the good level is reached, without modifying the term. Let `cPm4` be the rewrite rule `Pm4($x, $p)` ⇝ `Pm4($x, $p)`. This rule acts as the identity only for terms whose root is `Pm4`, and fails on any other ones. Thus, `ApplyOnceOpt(cPm4,arc1) = ITE(cPm4, arc1, One(ApplyOnceOpt(cPm4,arc1), 2))` performs the recursive descent only if `Pm4` has not yet been reached, avoiding the aforementioned unnecessary steps.

Another optimization consists in exploiting locality: transitions that affect the same places can be gathered so as to be evaluated together, thus avoiding unnecessary upwards and downwards walks of the state term. StrataGEM introduces the notion of *clusters* of places, a subset of the places of the net adjacent to a common set of transitions. In the Kanban model, `Pm2`, `Pback2` and `Pout2` together with transitions `tredo2`, `tback2` and `tok2` form such a cluster. These transitions can be applied at the cluster level, rather than on the whole term. The state terms slightly change to handle this optimization: the places of a given cluster are encoded contiguously in the state vector, limits of clusters being highlighted with new function names in the term. The transitions of a cluster are applied on the corresponding subterm only, identified by said cluster limits.

Beside avoiding unnecessary walks of the terms to find the appropriate level of application, we also note that the evaluation of a fixpoint can also be pushed at the cluster level. This is very similar to the optimization for fixpoint evaluation on Decision Diagrams known as saturation [2,8].

But the materialization of clusters also allows for further optimization. By an appropriate hierarchy in the state term and renaming of the places, cluster subterms can share their representation in memory. Thanks to DDs unicity tables and caches, this allows to share the representation and semantics of clusters with similar behaviors. This may considerably speed up the computations if the input net presents a high level of similarity between its clusters.

Details about the strategies generated for this example model, both non-optimized and optimized, are presented in `http://sourceforge.net/projects/stratagem-mc/files/` .

[1] This strategy also allows to define `Choice(S1, S2)` as `ITE(S1, Fail, S2)`, and `Sequence(S1, S2)` as `ITE(S1, S2, Fail)`.

Note that all these optimizations require to know (good) clusters for the input system. StrataGEM features a clustering algorithm, activated by default, that automatically detects clusters in the input net, and exploits them to use the above optimizations. The description of the clustering algorithm goes beyond the scope of this paper, but roughly speaking, it relies on a maximization of the ratio between the size of the cluster (the number of places it embeds) and its frontier (number of adjacent transitions). The automatic cluster detection is currently only available for Petri Nets, although similar algorithms could be designed for other input formats.

Sum Up We sum up in Table 1 the strategies available in StrataGEM. In this table, i and n are integers, `S`, `S1`, `S2` and `S3` denote strategies, `t` and $\mathtt{t}_1$ through $\mathtt{t}_n$ denote Σ-terms, `T` a set of Σ-terms, and `f` is a symbol of arity n in Σ.

Table 1. The strategies available in StrataGEM

simple strategy	wraps a term-rewriting rule as a strategy
one	$\texttt{One(S,}i\texttt{)[f(t}_1\texttt{,..., t}_n\texttt{)]} = \texttt{f(t}_1\texttt{,..., S[t}_i\texttt{],..., t}_n\texttt{)}$
choice	$\texttt{Choice(S1,S2)[t]} = \begin{cases} \texttt{S1[t]} & \text{if S1 is successful on t} \\ \texttt{S2[t]} & \text{otherwise} \end{cases}$
sequence	$\texttt{Sequence(S1,S2)[t]} = \begin{cases} \texttt{S2[S1[t]]} & \text{if S1 is successful on t} \\ \text{fails} & \text{otherwise} \end{cases}$
if-then-else	$\texttt{ITE(S1,S2,S3)[t]} = \begin{cases} \texttt{S2[S1[t]]} & \text{if S1 is successful on t} \\ \texttt{S3[t]]} & \text{otherwise} \end{cases}$
union	$\texttt{Union(S1,S2)[T]} = \begin{cases} \texttt{S1[T]} \cup \texttt{S2[T]} & \text{if S1 and S2 do not fail on T} \\ \text{fails} & \text{otherwise} \end{cases}$
fixpoint	$\texttt{Fixpoint(S)[T]} = \texttt{S}^n\texttt{[T]}$ where $n = \min_{i>0}(\texttt{S}^i\texttt{[T]} = \texttt{S}^{i+1}\texttt{[T]})$

3 Architecture

We shortly describe the architecture of the tool and its usage. StrataGEM can be used as a standalone command line tool or as a library. It is available at `http://sourceforge.net/projects/stratagem-mc/`, and we also provide a set of examples (including the Kanban model) at `https://sourceforge.net/projects/stratagem-mc/files/examples/`.

3.1 Architecture and Implementation

StrataGEM is written in Scala and comprises 3700 lines of code. It is packaged as a jar archive, as is common for Scala tools. The choice of Scala renders the tool platform-independent.

StrataGEM features three main parts, as shown on Figure 2. The core is the ΣDD implementation, with their relevant operations. It is extended by a layer implementing terms, rewrite rules, strategies and their combination

to describe a transition system. The third part is the import, that translates a model expressed in a given modeling language to a StrataGEM transition system whose semantics is described with rewrite rules and strategies. So far, StrataGEM features an import of Petri nets expressed in PNML [9]. This import includes an automatic detection of the clusters of a Petri net, as described above.

It also features an import from the language GAL [10]. This language serves as an intermediate, and already features translators from various inputs, such as DVE language used in the BEEM benchmark [11], amd Timed Automata, increasing the number of input formats for StrataGEM.

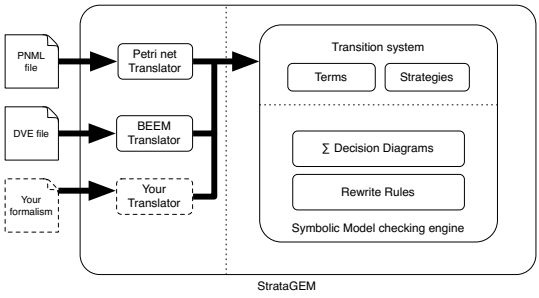

Fig. 2. Architecture of the StrataGEM

3.2 Standalone Tool

In its simplest form, StrataGEM currently computes the state space of the input model. Several levels of optimization are activated through the option `-t`, from the plain regular encoding described at the beginning of Section 2.2 to the use of hierarchized clusters as described at the end of Section 2.2.

The cluster detection, activated by default, is deactivated with an option. It can also be used alone, without computing the state space.

It is also possible to print the transition system with its rewrite rules and strategies generated for the input model by invoking `./stratagem analyzer -ts Kanban.pnml`.

Users also may add formalisms to StrataGEM. It only requires to translate the semantics of the input formalism into rewrite rules and strategies. The syntax of rewrite rules is quite straightforward and easy to use. All the strategies described in this paper are available to combine rewrite rules. It is also possible to define new strategies.

4 Benchmarks

We perform a quick experimental evaluation of our tool, to demonstrate the feasibility of our approach. Inspired by the Model Checking Contest @ Petri nets [12], we compare the performance of StrataGEM for the generation of the state space of some parametric Petri nets with the performance of PNXDD [13], a symbolic model-checker aimed at Petri nets. PNXDD also performs a cluster detection to produce hierarchized terms [14], and participated to the MCC since its first edition in 2011.

The compared tools present a lot of differences, so that the comparison of the raw data should be done very carefully. First of all, StrataGEM runs on the

Java virtual machine, that introduces an important time and memory overhead compared to PNXDD, written in C++. Moreover, PNXDD relies on a symbolic engine with years of maturity, highly optimized, whereas StrataGEM has no more than six months. The engine of PNXDD features many optimizations that are not at all present in StrataGEM. Also, StrataGEM was designed primarily as a demonstration prototype of the use of rewriting techniques on top of decision diagrams. Therefore, we choose to compare the *ratio* between the runtimes of both tools, so as to leverage these overheads. The evolution of the ratios among several instances of the same model gives us an indication of how this method scales compared to PNXDD.

We perform the comparison on three parametric models, expressed as Petri nets, used in the MCC 2013. The results are presented in Table 2. Besides the first, small, instances, models in the MCC are intended to be quite hard. That, and the current lack of fine-tuning of StrataGEM, limits the number of instances on which exploitable results can be obtained. An increase in the runtime ratio indicates that StrataGEM begins to close the gap with PNXDD. On the three models, we see that the ratio of runtime performance between StrataGEM and PNXDD evolves in favor of StrataGEM. The decrease of this ratio for the three first instances of the Eratosthenes model are anecdotic: the runtimes of PNXDD are close the time measurement errors on these instances. Furthermore, we are rather interested in the asymptotic behavior of this ratio. We also notice that StrataGEM ouperforms PNXDD on the last instance of the Railroad model, despite its aforementioned disadvantages.

Table 2. Runtime Comparison with clustering enabled

Model	Instance	PNXDD time (s)	StrataGEM time (s)	ratio P./S.
Eratosthenes	10	0.08	0.24	0.33
	20	0.11	0.79	0.14
	50	0.25	1.95	0.13
	100	1.14	6.48	0.18
	200	6.28	24.71	0.25
RailRoad	05	0.33	3.82	0.09
	10	103.90	83.06	1.25
SharedMemory	05	0.15	2.4	0.06
	10	1.08	3.55	0.30
	20	12.91	17.75	0.73

These preliminary results are quite encouraging, and are a first validation of the use of rewriting systems on top of symbolic data structures. As we have said, there is room for improvement and optimization in the StrataGEM prototype. Tuning work is planned in the near future, so as to make StrataGEM participate at the MCC 2014, that will enable a better assessment of its performance.

5 Conclusion

We have presented StrataGEM, a prototype implementing a novel approach for the manipulation of DDs. The operations on DDs are represented using rewriting rules and strategies to combine them. We have shown how they can describe the semantics of a Petri net. Other languages, such as the BEEM language, is also supported as an experimental feature. This approach is intended to address the need for a clear and user-friendly interface for the efficient manipulation of

such symbolic data structures. Our small assessment demonstrates that, despite its youth, our prototype has an acceptable performance. The room left for optimization and tuning promises quick improvements towards a participation in the next edition of the Model Checking Contest.

Beside this development work, future work includes the generalization of this approach to other formalisms. Algebraic Petri Nets seem to be a good middle-term target, and would demonstrate the flexibility of our approach.

References

1. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.: Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation 98(2), 142–170 (1992)
2. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation Unbound. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003)
3. Wolper, P., Godefroid, P.: Partial-Order Methods for Temporal Verification. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 233–246. Springer, Heidelberg (1993)
4. Buchs, D., Hostettler, S.: Sigma Decision Diagrams. In: Corradini, A. (ed.) TERMGRAPH 2009: Preliminary Proceedings of the 5th International Workshop on Computing with Terms and Graphs. Number TR-09-05 in TERMGRAPH Workshops, Università di Pisa, pp. 18–32 (2009)
5. Goguen, J.A., Meseguer, J.: Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. Theor. Comput. Sci. 105, 217–273 (1992)
6. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.E., Vittek, M.: ELAN: A Logical Framework Based on Computational Systems. Electronic Notes in Theoretical Computer Science 4, 35–50 (1996)
7. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: Piggybacking Rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007)
8. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical Set Decision Diagrams and Automatic Saturation. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 211–230. Springer, Heidelberg (2008)
9. International Organization for Standardization: ISO/IEC. Software and Systems Engineering - High-Level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation. International Standard ISO/IEC 15909 (December 2004)
10. MoVe Team: GAL, `http://move.lip6.fr/software/DDD/gal.php`
11. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
12. Kordon, F., Buchs, D.: Model Checking Contest Page, `http://mcc.lip6.fr/`
13. MoVe Team: The `PNXDD` Home Page (2013), `https://srcdev.lip6.fr/trac/research/NEOPPOD/wiki/pnxdd`
14. Hong, S., Kordon, F., Paviot-Adet, E., Evangelista, S.: Computing a Hierarchical Static Order for Decision Diagram-Based Representation from P/T Nets. In: Jensen, K., Donatelli, S., Kleijn, J. (eds.) ToPNoC V. LNCS, vol. 6900, pp. 121–140. Springer, Heidelberg (2012)