Building a Symbolic Model Checker from Formal Language Description

Edmundo López Bóbeda Maximilien Colange Didier Buchs
University of Geneva
Switzerland
{ edmundo.lopez, maximilien.colange, didier.buchs }@unige.ch

***Abstract*—The main limit towards practical model-checking is the combinatorial explosion of the number of states. Among numerous solutions proposed to tackle this problem, Decision Diagrams (DDs) have been proved efficient. They are however low-level data structures: translating a high-level model to them can be cumbersome. Indeed, little work towards their better usability has been undertaken.**

We propose an abstract mechanism for the manipulation of DDs, where system transitions are described in terms of rewrite rules. We describe how basic rewrite rules can be assembled through *strategies*, to describe complex transition relations (*e.g.* involving various levels of synchronization among parallel components). The strategies and rewrite rules offer a higher-level interface, where the nature of underlying DD is hidden, close to high-level languages used to model concurrent systems. We also describe specific strategies that we use to automatically translate high-level modeling languages (namely Petri Nets and imperative languages) to rewrite strategies, ultimately translated in terms of operations on DDs.

Keywords*—*symbolic model checking, term rewriting, semantics

I. Introduction

New languages and formalisms are born every year. Some of these languages need to provide verification features to their users. Ideally, these new languages should be able to reuse existing tools to offer these features without too much effort. Implementing a simulation engine for a formalism is usually not a complicated task. Nevertheless, performing a more thorough analysis almost always confronts the language developer with the difficulty of the semantics definition and the State Space Explosion (SSE) problem [1]. One of the techniques used to harness the SSE problem is symbolic model checking [2]. The problem thus reduces to reusing a symbolic model checker.

However, current symbolic model checkers are severely limited in the kind of models they can verify. Indeed, most symbolic model checkers can only handle models that use a predefined set of data types and operations, mostly integers and Booleans. This limitation is due to the underlying Decision Diagrams (DDs) and the lack of generality of the data type models they support.

This limitation makes the reusing of model checkers for languages other than their initial target almost impossible in practice. In fact, the translation of high level languages to

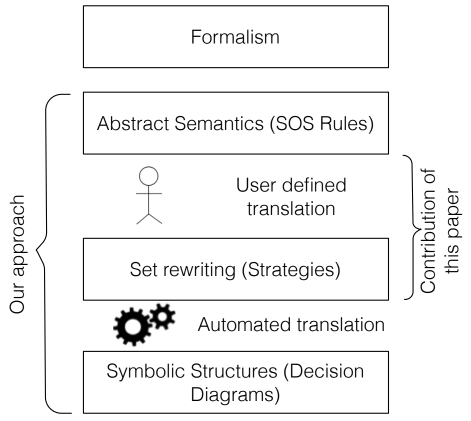
Formalism
Abstract Semantics (SOS Rules)
User defined translation
Set rewriting (Strategies)
Automated translation
Symbolic Structures (Decision Diagrams)
Our approach
Contribution of this paper

Fig. 1. Our approach and contribution

model checker low level formalisms presents several drawbacks: it can significantly increase model size, its correctness is hard to verify, and it is far from being straightforward. Two practical examples of these problems can be found in the Model-Checking Contest (MCC) [3]: the Drink Vending Machine and CSRepetition models. For the former, its textual representation as a Coloured Petri net (CPN) (high level) is only 24 Kb, while its textual representation as a Place/Transition Petri net (PN) is 139.9 Mb. The two versions of the latter model were proven to have different semantics where several tools reported different state spaces for them. This shows that even a translation that has been known for years and that presents no theoretical difficulty is still prone to errors.

Another popular approach translates the verification problem into a boolean SAT or SMT problem [4]. Verifiers thus easily benefit from improvements and breakthroughs in SAT/SMT solving. Nevertheless, the translation of data structures to SAT requires finite domains, which is a severe limitation of the method. Several years of research have demonstrated the efficiency of symbolic data structures to handle the SSE, and recent work [5] shows that it remains a serious competitor to SAT solving-based techniques. Also, SMT/SAT solvers are limited in general to integers and Boolean data types.

Rather than using an existing symbolic model checker or SAT/SMT solver, one can also write a translation directly to the data structure used for symbolic model checking, *i.e.,* DDs [6], [7], [8], [9]. The problem here is almost the same as with translation to model checkers. The data types supported by most DDs are integers and Booleans. A notable exception are ΣDDs [7], which support custom defined operations and data types based on Abstract Data Types (ADTs). However, their usage in model checking was limited to the rewriting of sets of terms in the model checking of Algebraic Petri nets (APNs) [10].

Finally, whichever approach is chosen to implement the model checker, the user has to deal with optimizations. Model checkers usually provide such optimizations for free for their primary targetted formalisms. However, information relevant to optimization is sometimes "lost in translation". Taking advantage of model checker built-in optimizations complicates even more the reuse of an existing model checker.

A lot of work has been undertaken to raise the level of abstraction of DDs, from Binary Decision Diagrams (BDDs), based entirely on Boolean functions, to recent ΣDDs, manipulating ADTs to handle high level specifications. This evolution is detailed in Section VII. Nevertheless, translating a high level model to ΣDDs still requires a comprehensive knowledge of the DD structure.

We propose a framework to abstract the DD-specific parts. On one hand this helps to ease the translation process. On the other hand it allows further research to be focused specifically on DDs. To tackle the aforementioned problems our framework systematizes the construction of a symbolic model checker. Our framework, shown in Figure 1, comprises three layers. Each layer describes the semantics of the formalism at a different level of abstraction. The upmost layer is the denotational semantics of the formalism expressed as functions on mathematical objects. It represents an abstract view of the semantics. A concrete semantics in this layer is given by Structured Operational Semantics (SOS) rules [11]. The middle layer represents the operational semantics using Set Rewriting (SR). The building blocks of a SR-based semantics are strategies. SR is our core contribution and is fully introduced in Section III. It allows to express semantics, optimizations and model checking computations. It shows a concrete view of the semantics and can easily be translated to a very fast and memory efficient representation for model checking. This translation is omitted from this paper for space reasons, we detail it in [12]. The bottom layer is based on ΣDDs. Actual computations are done in this layer. We provide a working implementation of the strategies for the middle layer, the translation to the lower layer, and the ΣDD-based lower layer. The user only has to provide the Term Rewriting (TR) strategies (in Figure 1, the "User-defined translation") in order to automatically have an efficient symbolic model checker for the source formalism. The main contribution of this paper is the introduction of SR together with a systematic approach to produce the "User-defined translation" from Figure 1.

Outline Section II first presents the abstract semantics of a concurrent system. Section III and Section IV are at the heart of our contribution, introducing set rewriting, and its usage to represent the semantics. Section V sketches the DDs used for implementation. Section VI proposes an experimental assessment. Finally, related work is discussed in Section VII, and Section VIII concludes the paper.

II. Abstract semantics

We choose to describe the abstract semantics with the well-known SOS rules. The semantics in this form is given as a set of inference rules called SOS rules. Each rule, based on predicates defined on semantic domains, provides a way to deduce the existence of a transition in the semantics. A set of SOS inference rules, *i.e.,* rules satisfying certain constraints, produces a sound transition system.

We consider only SOS rules of the form

$$\frac{c}{s \to a(s)} \tag{1}$$

where $s \in \Gamma$ is the state from the state space Γ, $c \in \mathit{Cond}$ is a condition, and $a \in \mathit{Act}$ is a function that transforms a state to another state. The arrow $\to$ represents a visible transition in the transtion system.

For languages with a more imperative structure, SOS rules may be presented differently. For example one can have rules that contain sub-transitions in the numerator. We do not consider such rules, but the translation method we provide later allows to represent equivalent semantics with only simple rules.

For example, Petri nets (PNs) have a very simple semantic definition based on one inference rule. Formally, a PN is an n-tuple $\langle P, T, \mathit{in}, \mathit{out}, m_0\rangle$ where P is a finite set of places, T is a finite set of transitions, and in and out are respectively the pre and post conditions (in curried form) $\mathit{in} : T \to (P \to \mathbb{N})$ and $\mathit{out} : T \to (P \to \mathbb{N})$. The marking (or state) is a mapping from places to integers: $m : P \to \mathbb{N}$. Each transition t yields a semantical rule:

$$\frac{m \geq \mathit{in}(t)}{m \to_t m - \mathit{in}(t) + \mathit{out}(t)} \tag{2}$$

where $m, m_1, m_2 \in (P \to \mathbb{N})$, $m_1 \geq m_2$ is true if and only if $\forall p \in P \ : \ m_1(p) \geq m_2(p)$, and $(m_1 + m_2) : p \mapsto m_1(p) + m_2(p)$. SOS rule (2) clearly corresponds to the template presented by Rule (1).

III. Set Rewriting Strategies

In this section we present the core of our contribution, namely a language to describe operations on DDs and its semantics. The semantics of this language is given over sets of terms. Its building blocks are Set Rewriting (SR) strategies. Our ultimate goal is to translate a semantics given in SOS rules to a semantics described using SR. Basically SR needs a way to represent the states and a way to describe a transition, *i.e.,* the SOS rule. In SR one describes states using terms, and SOS rules using SR strategies. In the same way that a SOS rule describes how to compute the successor of a given state by some transition, SR describes how to compute the set of successors of a set of states. This section is divided in two subsections. We first give the formal definition of Term Rewriting (TR) in Section III-A, which can be skipped by the familiar reader. We then describe our SR strategies, which bring some original definitions.

$$R = \left\{ \begin{array}{l} 1.\ \texttt{not(true)} \rightsquigarrow \texttt{false} \\ 2.\ \texttt{not(false)} \rightsquigarrow \texttt{true} \\ 3.\ \texttt{and(true, \$x)} \rightsquigarrow \texttt{\$x} \\ 4.\ \texttt{and(false, \$x)} \rightsquigarrow \texttt{false} \end{array} \right\}$$

Fig. 2. Boolean rewrite rules

A. Term Rewrite Systems

The theory in this section was defined in [12] with the more complete Order-Sorted Signature [13], limited here to Many-Sorted Signature (MSS) [14] for simplicity. For a formal language, a MSS lists the data types (*sorts*), and the symbols available in the language (*function names*). The sentences of a language are generated from its MSS.

A Many-Sorted Signature is a pair $\Sigma = \langle S, F\rangle$, where S is a finite set of sorts and $F = (F_{\omega,s})_{\omega\in S^*, s\in S}$ is a $(S^* \times S)$-sorted set of function names. Every $f \in F_{\epsilon,s}$ is called a *constant*, with ϵ the empty string.

Each sentence in the formal language described by a MSS is called a *term*. Let $\Sigma = \langle S, F\rangle$ be a MSS and X be an S-sorted set of variables. The set of terms over Σ with sort $s \in S$, denoted by $(\mathcal{T}_{\Sigma,X})_s$, is the least set inductively defined by:

- $x \in (\mathcal{T}_{\Sigma,X})_s$ for all $x \in X_s$, and $c \in (\mathcal{T}_{\Sigma,X})_s$ for all $c \in F_{\epsilon,s}$;
- $f(t_1, \ldots, t_n) \in (\mathcal{T}_{\Sigma,X})_s$ for all $f \in F_{s_1,\ldots,s_n,s}$, and for all $t_i \in (\mathcal{T}_{\Sigma,X})_{s_i}$ with $1 \leq i \leq n$.

Composite terms, *i.e.,* not reduced to a constant or a variable, consist of a function symbol concatenated to terms of compatible sorts, called subterms.

The operational semantics of a system is defined through successive rewritings of the term until a final value is reached. A rewrite step is described by *rewrite rule* (or TR rule), an ordered pair of terms $l \rightsquigarrow r \in \mathcal{T}_{\Sigma,X} \times \mathcal{T}_{\Sigma,X}$. A set of rewrite rules is a Term Rewrite System (TRS). For example, the signature of Booleans consists of a single sort B, two constants true and false, a single unary function symbol not and a single binary function symbol and. Figure 2 shows TRS for this MSS. Please note that variables (in rules 3 and 4) are prefixed by a dollar sign $.

A substitution $\theta : X \mapsto \mathcal{T}_{\Sigma,X}$ is a set of variable assignments. θ naturally extends from variables to terms: in postfix notation, $t\theta$ is the term t where each variable x has been replaced by $x\theta$. For the substitution $\theta = \{x \mapsto false\}$, we have $\mathrm{and(true,\ x)}\theta = \mathrm{and(true,\ false)}$.

A term s rewrites to t by some rewrite rule $l \rightsquigarrow r$ in R (also noted $s \xrightarrow{\langle l,r\rangle,C} t$) if there is a term C with a single occurrence of the fresh variable X, and a substitution θ such that $s = C\{X \mapsto l\theta\}$ and $t = C\{X \mapsto r\theta\}$. If we have $C = X$ we say that the rules is applied at the *root* of the term. For the term $t = \mathrm{and(true,\ not(false))}$, there are two possible rewrite steps from the rules in Figure 2: $t \rightarrow_R \mathrm{not(false)}$ by rule 3, and $t \rightarrow_R \mathrm{and(true,\ true)}$ by rule 2.

B. Strategies

Strategies are at the heart of Set Rewriting (SR), and are inspired from the strategy languages of Maude [15], Tom [16] and ELAN [17]. We use strategies to describe the different transitions of a system. Each strategy may represent either one or several transitions. The main particularity of our approach is that our strategies are defined on sets of terms, and not only terms. In particular, given a set of states (expressed as terms), a strategy operationally describes how to obtain the set of successor states of the given set. We also introduce two brand new strategies, namely the `Union` and `Fixpoint` strategies. Apart from describing operational semantics, our strategies have two secondary goals to improve their model checking capabilities. First, we want to completely remove non determinism for the application of rewrite rules. Removing non determinism allows to have a faster implementation and makes reasoning on rewrite rules easier. Second, we want to control how rules are applied. This allows us to tune the computation described by the strategy to make it even faster.

A strategy is a partial function $Strat : \mathcal{P}(\mathcal{T}_{\Sigma,X}) \nrightarrow \mathcal{P}(\mathcal{T}_{\Sigma,X})$[1]. If *defined* on a set of terms T, a strategy applied to T yields a new set of terms. We say that a strategy *fails* on sets of terms on which it is not defined. Our framework provides means to describe such partial functions using rewrite rules, and to combine them using predefined strategy operators. These are themselves strategies, *i.e.,* they can be again combined to create even more complex strategies. The users can also create their own combination operators. Below we introduce the different strategies of our framework.

A *simple strategy* is the SR equivalent of a TR rule. Whereas a TR rule describes a local transformation of a term, an SR simple strategy describes a local transformation on a *set of terms*. A simple strategy consists of an ordered list of TR rules. Intuitively its application proceeds as follows: in the ordered list, pick the first rule that can rewrite at least one term of the input set. If no such rule exists, then the whole strategy fails; otherwise, the picked rule is applied to all the terms of the input set. The result of the operation is the set containing all terms that could be rewritten by the picked rule. Note that in our setting the application of the TR rule is always done at the root of the terms. More formally:

$$\{\mathtt{r}_1, \cdots, \mathtt{r}_n\}[T] = \begin{cases} \textit{fail} & \text{if no rule in } \{\mathtt{r}_1, \cdots, \mathtt{r}_n\} \\ & \text{can be applied} \\ \textit{Rew}_{\mathtt{r}_k}(T) & \text{if } k \text{ is minimal s.t. } \mathtt{r}_k \\ & \text{is defined on the set } T \end{cases}$$

where the $\mathtt{r}_i$'s are rewrite rules, $T \in \mathcal{T}_{\Sigma,X}$ is a set of terms, and $\textit{Rew}_r(T) = \left\{ t | s \in T \wedge s \xrightarrow{r,X} t \right\}$ with $s \xrightarrow{r,X} t$ denoting the rewriting of s to t with rule r applied at root.

To describe operations that modify subterms (instead of only the root of the term), we use the $\mathtt{One(S)}$ or the $\mathtt{One}_k\mathtt{(S)}$ strategy. The $\mathtt{One(S)}$ strategy is a parametric strategy that takes one parameter. It is defined in terms of the $\mathtt{One}_k$ strategy. Intuitively, the application of $\mathtt{One}_k\mathtt{(S)}$ to a set of terms T amounts to the application of S to the set T' of all the k-th subterms of terms of T. If S fails on T', then $\mathtt{One}_k\mathtt{(S)}$ fails on T. Otherwise, each element of $\mathtt{S}(T')$ was, before rewriting, a subterm of an element of T; the final result of the strategy is a set containing all elements of T with their original k-th subterm replaced by the rewritten element in $\mathtt{S}(T')$. In practice,

[1]We use the notation $\nrightarrow$ to differentiate partial function from total functions

$$
\begin{aligned}
\mathit{ITE}(S_1,S_2,S_3)[T] &= \begin{cases} S_2[S_1[T]] & \text{if } S_1 \text{ succeeds} \\ S_3[T] & \text{if } S_1 \text{ fails} \end{cases} \\
\mathit{Sequence}(S_1,S_2)[T] &= \mathit{ITE}(S_1,S_2,\mathit{Fail})[T] \\
\mathit{Choice}(S_1,S_2)[T] &= \mathit{ITE}(S_1,\mathit{Identity},S_2)[T] \\
\mathit{Union}(S_1,S_2)[T] &= \begin{cases} \mathit{fail} & \text{if } S_1[T] \text{ or} \\ & S_2[T] \text{ fails} \\ S_1[T] \cup S_2[T] & \text{otherwise} \end{cases} \\
\mathit{Identity}[T] &= T \\
\mathit{Fail}[T] &= \mathit{fail} \\
\mathit{Not}(S)[T] &= \begin{cases} \mathit{fail} & \text{if } S[T] = T \\ T & \text{if } S[T] \text{ fails} \\ T \backslash S[T] & \text{if } S[T] \text{ succeeds} \end{cases} \\
\mathit{FixPoint}(S)[T] &= \begin{cases} T & \text{if } S[T] = T \\ \mathit{FixPoint}[T'] & \text{if } S[T] = T' \end{cases}
\end{aligned}
$$

Fig. 3. Semantics of Set Rewriting strategies

all of this book keeping is done by the ΣDD structure. Its formal semantics is the following (note that $\texttt{One}_k\texttt{(S)}$ cannot return an empty set, it would fail instead):

$$
\texttt{One(S)}[T] = \begin{cases} \texttt{One}_k\texttt{(S)}[T] & \text{if } k = \min(1,\cdots,n) \text{ s.t.} \\ & \texttt{One}_k\texttt{(S)}(T) \neq \mathit{fail} \\ \mathit{fail} & \text{otherwise} \end{cases}
$$

$$
\texttt{One}_k\texttt{(S)}[T] = \begin{cases} \{f(t_1,\cdots,t',\cdots,t_n) \mid \\ \quad f(t_1,\cdots,t_k,\cdots,t_n) \in T \text{ and } t' \in \texttt{S}(t_k)\} \\ \mathit{fail} \quad \text{if the above set is empty} \end{cases}
$$

Simple strategies along with `One` are at the core of SR. With them it is possible to describe local transformations at any position of a term. These strategies have a direct translation to ΣDDs. The rest of the strategies are combination strategies and are depicted in Figure 3. Using the combination strategies one can describe a global transformation, *e.g.,* apply a set of local transformations atomically. The `Sequence` strategy can be used for that as well as to chain two strategies, handling the case when they are not defined. The `Union` strategy allows to reintroduce the non-determinism of the semantics in our framework. The `Fixpoint` strategy describes fixpoint computations, usually smallest fixpoints so as to ensure convergence. It is for instance used to compute the whole state space, as a transitive closure of the transition relation. The `ITE` strategy is a general operation to conditionally apply strategies. It subsumes `Sequence`, and is typically used to describe optimization for the evaluation of the rewrite rules.

IV. From SOS Rules to Set Rewriting Strategies

In this section we present a systematic way to translate SOS rules to strategies. It is worth noting that there are approaches to translate SOS rules to rewrite rules [18]. However, to the best of our knowledge there is no research concerning the transformation of SOS rules to standard rewrite strategies. Besides, our strategy language has certain particularities that render our approach unique due to its application on sets of terms. In this section we take a pedagogical approach. First we present a general approach to translate a semantics given with SOS rules. Then we study the translation of two formalisms: Petri nets and Divine [19] (a SPIN-like [20] formalism).

A. General approach

We assume that the goal of the user is to have a working symbolic model checker for models of his formalism. The typical user of our framework starts with two things: a formalism, which has a syntax and a semantics (SOS rules), and instances of the formalism (we call these models). To achieve this goal the user must provide what we call the "User-defined translation" (the *translation*). The translation transforms a model of the given formalism to an SR model. The SR model consists of three things: a signature (cf. Section III-A), an initial state, and a set of transitions expressed as strategies. Since SR also allows the definition of new strategies, an SR transition system also supports the definition of new strategies. We omit such definitions from the translation. However, we clearly indicate each time a new strategy is being defined. In practice, the translation consists of two functions: 1) a function *compState* ("compile state") that translates a model to a signature Σ and an initial state $t_0 \in \mathcal{T}_\Sigma$, and 2) a function *comp* ("compile") that translates model transitions to SR transitions.

The *compState* function is easy to derive. Indeed, one only needs to recreate the data structure that contains a state using terms. This structure is usually a map from variable names to some data type. For the sake of brevity and because of its simplicity, we omit a general description of *compState* function. We present only the description of the states for the case studies in the next subsections.

Before defining the *comp* function we give a more formal structure to the simple SOS presented in Rule (1). We say that a function returning a Boolean is a predicate. We then define the set Cond inductively: If pred is a predicate, then $\mathit{pred} \in \mathit{Cond}$. Also, if $p, p' \in \mathit{Cond}$, then $p \wedge p' \in \mathit{Cond}$. Finally, if a is a function such that $a \circ p$ is a predicate, then $a \circ p \in \mathit{Cond}$. Similarly we define the set Act. If $\mathit{act} : \Gamma \longrightarrow \Gamma$ is a function that transforms a state to another, then $\mathit{act} \in \mathit{Act}$. Also, if a, a' are functions such that their composition $a \circ a' : \Gamma \longrightarrow \Gamma$ then $a \circ a' \in \mathit{Act}$.

The *comp* function takes SOS rules (representing a transition) and returns an equivalent strategy. As stated before we treat only the case of SOS rules of the format shown in Rule (1) (we call them *simple SOS rules*). To overcome the limitation of simple SOS rules we proceed in three stages. First we describe formally the different predicates and actions, *i.e.,* the predicates in the set Cond and the functions in the set Act. Second, we craft the simple SOS rules from the functions and predicates we defined. Finally, we provide the specific translation for the predicates and functions to strategies. The general translation is given below. We provide placeholder functions for the translation of the formalism specific parts.

Let us formally describe the *comp* function. The *comp* function obviously depends on the formalism being compiled. That dependency is visible in the *userPredComp* and *userActComp* functions. These two functions are specific to each formalism. Here we factor out the specific parts and

nat	zero: nat s: nat $\mapsto$ nat
pname	P1: pname P2: pname $\vdots$
mapping	empty: mapping map: pname, nat, mapping $\mapsto$ mapping

TABLE I. TERM SIGNATURE FOR PETRI NET MARKINGS

present a very general function.

$$\begin{aligned} comp &: \frac{c}{s \to a(s)} \mapsto \texttt{Sequence}(comp(c), comp(a)) \\ &: c \wedge c' \mapsto \texttt{Sequence}(comp(c), comp(c')) \\ &: a \circ a' \mapsto \texttt{Sequence}(comp(a), comp(a')) \\ &: a \circ p \mapsto \texttt{Sequence}(comp(a), comp(p)) \\ &: pred \mapsto userPredComp(pred) \\ &: act \mapsto userActComp(act), \text{ if } act : \Gamma \to \Gamma \end{aligned}$$

This definition relies on a few assumptions. First, the strategy resulting from compiling a condition, *i.e.,* the strategy $comp(c)$, must fail if the condition is not satisfied by any term of the input set. This ensures that if a condition in a SOS rules does not hold then its corresponding strategy fails. Second, if the condition strategy succeeds on input T resulting in T', then T' must be the subset of T that satisfies condition c. Third, the compilation of an action guarded by c needs to be always defined on all the inputs that pass condition c.

In the following case studies we show examples of the definition of both formalism specific functions. This includes also the translation of concrete predicates and actions to strategies. In the following case studies we treat strategies and rewrite rules as if they worked on single terms. For most cases, reasoning on terms greatly simplifies the creation of a translation and is also correct. Each time that the single term reasoning does not apply we precise accordingly.

B. Case Study: Petri nets

This translation is implemented in our tool Strategy Generic Extensible Modelchecker (StrataGEM) [21].

1) The intermediate language: Given the set of places of one PN we define its set of states as the set $\Gamma = \{m \mid m : P \to \mathbb{N}\}$ of all potential markings. We first define one predicate $test_{p,n} : \Gamma \longrightarrow \mathbb{B}(= \{true, false\})$ to test if a place p in a given state contains at least n tokens. We also need an action to decrease the number of tokens in a place and another one to increase the number of tokens in a place. We propose the following functions: $dec_{p,n}$ and $inc_{p,n}$, where p is a place and n a strictly positive integer, both functions are of type $\Gamma \longrightarrow \Gamma$. We can now define the simple SOS rule corresponding to a PN transition as follows:

$$t = \frac{true = \bigwedge_{p \in P} test_{p,in(t)(p)}(m) \quad P = \{p_1, \ldots, p_{n'}\}}{m \to a(m)}$$

with $a = dec_{p_1,in(t)(p_1)} \circ \cdots \circ dec_{p_{n'},in(t)(p_{n'})} \circ inc_{p_1,in(t)(p_1)} \circ \cdots \circ inc_{p_{n'},in(t)(p_{n'})}$ (semantically, the order of application does not matter).

2) Describing the state: In our framework states are simple terms: the set of states of a PN is the set of its markings (*i.e.,* mappings from P to $\mathbb{N}$, see Section II). We propose the signature described in Table I. There is a sort for natural numbers, a sort for places and a sort representing a map. Natural numbers are encoded as terms being either `zero`, or `suc(t)` where `t` is an integer term. For clarity, integer terms are often replaced by plain integers. For instance, the initial state of a PN with three places is `map(P1, 0, map(P2, 5, map(P3, 0, empty)))`. `mapping` is an inductive structure, which allows to define generic strategies to read and modify it. The same structure can be used for a system whose states represent sets of variable assigments.

3) The comp function: To complete the definition of $comp$, we proceed to define the $userPredComp$ and $userActComp$ functions. Our goal is to define simply how a specific predicate or action is written as a strategy.

We start by translating the predicate $test_{p,n}$. Our testing strategy first locates the place p thanks to a parametric strategy `applyTo(checkLoc, S)` = `ITE(checkLoc, S, One3(applyTo(checkLoc, S)).` The `applyTo` strategy takes two parameters: a all-or-nothing strategy `checkLoc` to select the appropriate location, and a strategy `S` to be applied at the position selected by `checkLoc`. The `applyTo` strategy first applied `checkLoc`. If it fails, we have not reached the desired location, and `applyTo` applies itself recursively on subterms. When the appropriate depth is reached, `checkLoc` succeeds returning its input set unchanged, and `S` is applied. Thus, our translation is done by creating sound `checkLoc` and `S` strategies.

Since for a given PN we know all the places, we can directly define a $\texttt{checkLoc}_p$ strategy for each place p: $\texttt{checkLoc}_p$ = `{ map(p, $x, $m)` $\rightsquigarrow$ `map(p, $x, $m) }`[2]. Given a set of terms, the strategy $\texttt{checkLoc}_p$ returns only the subset of elements whose first subterm contains the place p. In particular it fails if no term in the set contains the place p at the first subterm. We then propose $\texttt{testCond}_n$ = `{ map($p,` $compInt(n)$ `, $m)` $\rightsquigarrow$ `map($p,` $compInt(n)$ `, $m)` that performs the actual test. The function $compInt : \mathbb{N} \longrightarrow \mathcal{T}_{\Sigma,X}$ takes a natural number n and returns a term capable of matching a larger or equal number. Its definition is given below:

$$\begin{aligned} compInt &: \mathbb{N} \to \mathcal{T}_\Sigma \\ &: 1 \mapsto \texttt{s(\$x)} \\ &: n \mapsto \texttt{s(}compInt(n-1)\texttt{)} \end{aligned}$$

Using the aforepresented functions we can define the $userPredComp$ function as follows:

$$userPredComp : test_{p,n} \mapsto \texttt{applyTo(checkLoc}_p\texttt{, testCond}_n)$$

Finally we need to define the $userActComp$ strategy, based on the translation of our $dec_{p,n}$ and $inc_{p,n}$ functions. The modularity of our framework allows to reuse the strategies we

[2]For simplicity the term `p` corresponds to the place p. Please note that `p` is not supposed to be a variable (in the rewriting sense) but a constant representing a place.

already defined, thus we proceed directly to outline the form of the $userActComp$ function:

$$\begin{aligned} userActComp &: dec_{p,n} \mapsto \texttt{applyTo(checkLoc}_p\texttt{, decStrat}_n\texttt{)} \\ &: inc_{p,n} \mapsto \texttt{applyTo(checkLoc}_p\texttt{, incStrat}_n\texttt{)} \end{aligned}$$

Finally this definition only leaves us with the definition of a translation for `decStrat`$_n$ and `incStrat`$_n$. Let us consider `decStrat`$_n$ = `{ map($p,` $compInt(n)$ `, $m)` $\rightsquigarrow$ `map($p, $x, $m)}`. This strategy simply removes n tokens from `p`. `incStrat`$_n$ presents exactly the same structure, only that the terms on the rule are inverted.

C. Case Study: Divine

This translation is also implemented in our tool StrataGEM. Divine is a language whose syntax resembles that of SPIN [20]. The Divine formalism is aimed at modeling concurrent systems. A Divine model features a set of processes and global variables. Each process defines a set of internal variables and transitions. Each transition might read and write its owner process's variables as well as the global ones. For the sake of brevity we only present formal definitions on a need-to-know basis. For the moment we consider a Divine instance as a tuple $\langle V, P, (V_p)_{p\in P}\rangle$, where V is the set of global variables, P the set of processes, and $(V_p)_{p\in P}$ a family of sets indexed by the processes, each set represents the local variables of a process. We assume that for all $p, p' \in P$ $(p \neq p')$, we have $V_p \cap V_{p'} = V_p \cap V = \emptyset$. Also, without loss of generality we suppose that variables always store integers $i \in \mathbb{Z}$. Also, we present here only a simplified version of Divine. We focus mainly on transitions.

The set of expressions $Expr$ is inductively defined as the smallest set containing all integers, all variables of the Divine model, and additions and substractions thereof (we limit to two operators for the sake of brevity). An assignment is a pair $v := e$ were v is a variable and e an expression. We also consider the set of Booleans $\mathbb{B}$. A guard is a boolean combination of comparisons (only equality here for simplicity) between expressions.

A transition in Divine consists of a guard expression followed by a list of variable assignments. Formally we write a transition as $\textbf{trans}\ g; as \in Trans$, where $g \in Guards$ and $as \in Assign$.

1) The intermediate language: As we did with PNs, we proceed to describe the actions and predicates that form the building blocks of our SOS rules. We propose an intermediate language working on a stack machine, *i.e.,* a machine that uses a pushdown stack instead of machine registers. The state of the machine is represented by a tuple $\langle st, m\rangle$, where $st \in Stack \cup Guards$ is a stack of integers (or a guard expression), and $m \in M = \left\{m \mid m : V \cup \bigcup_{p\in P} V_p \longrightarrow \mathbb{N}\right\}$ is a map from variables to natural numbers. Thus, we define a the set of a states of a Divine model as a $\Gamma = (Stack \times M) \cup (Guards \times M)$.

The intermediate language is depicted completely in Figure 4. Machine stack operations have the following semantics: $push$ (do not confuse with $push_n$ which is an instruction of the intermediate language) adds an element to the stack and returns the stack, pop removes the top element of the new stack and returns the stack, and top returns the top element of the stack.

$$\begin{aligned} test &: BoolExpr \times M \to \{true, false\} \\ &: \langle b, \rho\rangle \mapsto \begin{cases} true, \text{ if } b = true \\ false, \text{ otherwise} \end{cases} \\ push_n &: Stack \times M \to Stack \times M \\ &: \langle st, \rho\rangle \mapsto \langle push(n, st), \rho\rangle \\ add &: Stack \times M \to Stack \times M \\ &: \langle st, \rho\rangle \mapsto \langle push(top(st) + top(pop(st)), pop(pop(st))), \rho\rangle \\ subt &: Stack \times M \to Stack \times M \\ &: \langle st, \rho\rangle \mapsto \langle push(top(st) - top(pop(st)), pop(pop(st))), \rho\rangle \\ read_v &: Stack \times M \to Stack \times M \\ &: \langle st, \rho\rangle \mapsto \langle push(\rho(v)), \rho\rangle \\ write_v &: Stack \times M \to Stack \times M \\ &: \langle st, \rho\rangle \mapsto \langle pop(st), \rho[v = top(st)]\rangle \\ eq &: Stack \times M \to BoolExpr \times M \\ &: \langle st, \rho\rangle \mapsto \langle top(st) = top(pop(st)), \rho\rangle \end{aligned}$$

Fig. 4. Semantics of the Intermediate language

Using the operations in Figure 4 one can describe the semantics of Divine using simple SOS rules. The intermediate language is a set of functions and consequently they are composed according to their profile to produce a pre-condition and a post-condition in the SOS rule.

Our next step is to provide a description of a Divine transition by a simple SOS rule. We present such a function in Figure 5. The function creates a simple SOS rule for each Divine transition.

The translation of $\textbf{trans}\ g; a$ is :

$$\frac{true = lComp(g)(\langle st, m\rangle)}{\langle st, m\rangle \to lComp(a)(\langle st, m\rangle)}$$

which has the same shape as the rules already presented. The complexity is hidden in the compilation function.

To illustrate the working of this function we present an example. Let us consider a system with two global variables $v, v' \in V$ and translate the following transition: $\textbf{trans}\ v = 1 + v'\ ;\ v := v + 1$. The reader can verify that the corresponding simple SOS rule is the following:

$$\frac{true = test \circ eq \circ read_{v'} \circ add \circ read_v \circ push_1(\langle st, m\rangle)}{\langle st, m\rangle \to write_v \circ add \circ read_v \circ push_1(\langle st, m\rangle)}$$

This rule describes exactly what is happening to the stack and memory after the evaluation of this Divine instruction. One may argue that some sub-expressions are similar on the top and in the bottom and then produce inefficiencies. It must be noted that this is automatically optimized by the underlying operational mechanisms based on memoization.

2) Describing the state: This set of states is almost exactly the same as the one we used for PNs. Thus we encode it using the signature presented in Table I. We also need to encode the stack. To avoid the declaration of new data structures we encode the stack using a new variable on top of the existing map. We also intentionally omit a declaration for the integer sort and refer the reader to the examples available for our tool. For example, given the term `m` that represents a map, and an integer `n`, we represent the pair map integer with the following term: `map(t, n, m)`.

Concerning the Boolean expressions that we allow in the state, we extend the signature with the Boolean type

$$\begin{aligned}
\mathit{lComp} &: \mathit{DivBasicExpr} \to \mathit{Cond} \cup \mathit{Act}\\
&: c \textbf{ and } c' \mapsto \mathit{lComp}(c) \wedge \mathit{lComp}(c)\\
&: \mathit{true} \mapsto \mathit{true}\\
&: \mathit{false} \mapsto \mathit{false}\\
&: e = e' \mapsto \mathit{test} \circ \mathit{eq} \circ \mathit{lComp}(e) \circ \mathit{lComp}(e')\\
&: e + e' \mapsto \mathit{add} \circ \mathit{lComp}(e) \circ \mathit{lComp}(e')\\
&: e - e' \mapsto \mathit{subt} \circ \mathit{lComp}(e) \circ \mathit{lComp}(e')\\
&: \mathit{id} := e \mapsto \mathit{write}_v \circ \mathit{lComp}(e)\\
&: a; a \mapsto \mathit{lComp}(a) \circ \mathit{lComp}(a)\\
&: n \mapsto \mathit{push}_n\\
&: v \mapsto \mathit{read}_v
\end{aligned}$$

Fig. 5. Translation from Divine to simple SOS rule

and we add an operator `test: bool, mapping` $\mapsto$ `state`. The Boolean type contains the constants: `true` and `false`. It also contains the comparison operator `eq: nat, nat` $\mapsto$ `bool`. Using these types we can now define the *comp* function. We also define operations in the signature, such as `+: nat, nat` $\mapsto$ `nat` and `-: nat, nat` $\mapsto$ `nat`. We call terms containing one of such operation (plus, substraction, eq) *operation terms*.

3) The comp function: Given our intermediate language, we only need to provide the *userPredComp* and *userActComp*. For *userPredComp* we only have one translation to do, *i.e.,* the *test* predicate. The translation is straightforward because most of the work is done by the other functions. The translation is given below:

$$\mathit{userPredComp} : \mathit{test} \mapsto \texttt{test(true, \$m)} \rightsquigarrow \texttt{\$m}$$

This translation assumes that the functions that are applied before the test reduce the conditions to either `true` or `false`. Then when we apply the aforementioned rule to a set of terms we obtain the desired behavior. The rule returns only the terms where the condition is true (recall that the rule is applied to a set of terms) and fails if no term can be rewritten by it.

Next we treat the translation of actual operations, in our case operations on integers. We have defined three operations in Figure 4: *add*, *subt*, and *eq*. The principle for treating operations is always the same. We need a set of rules that performs the actual transformation and a strategy that ensures that the rules are applied correctly to the set of terms. We present the two parts for the *eq* translation. First, we define the actual rules using the following simple strategy:

```
equals = {eq(0, 0) ⇝ true,
      eq(suc($n1), suc($n2)) ⇝ eq($n1, $n2),
      eq(0, suc($n1)) ⇝ false,
      eq(suc($n1), 0) ⇝ false }
```

The next step is to define the strategies that apply the `equals` strategy. We assume that the set of terms on which the `equals` strategy is applied does not contain operation terms. Given a set of terms T, where for each $t \in T$ we have t of the form `test(eq(`t_1`, `t_2`), `t_3`)`, we want to reduce it to a set of terms T'. The new set T' contains only elements of the form `test(true, `t_1`)` or `test(false, `t_1`)`. Thus, we need to define an `applyEquals` strategy to accomplish this result. One might argue that the semantics of the simple rule cannot accomplish the task because it removes all elements that were not rewritten. Thus we need to declare a new strategy:

```
RewriteSet(S) = Choice(Union(S, Not(S)),
                Choice(S, Not(S)))
```

The `RewriteSet` strategy enables to use SR rule without removing the terms that were not rewritten. To do this, it uses the `Not` strategy (cf. Figure 3 for its semantics). Its workings is easy to explain. It tries to do the union of the SR simple strategy with the its complement (*i.e.,* the `Not` strategy). If one of the two fails, then it tries to execute only one of them. Finally, we can present the a strategy that applies the `equals` strategy obtaining the desired result:

$$\texttt{applyEquals} = \texttt{One}_1\texttt{(Fixpoint(RewriteSet(equals)))}$$

The strategies for handling the other operations all follow the same pattern presented for the `eq` operator. The only difference is the simple strategy. The rules contained in the simple strategies are the same rules that one would use in a classical TR setting.

Finally, we need to describe the read_v and write_v translations. We present here the strategy `readV`. It reads a value from the map and then puts it on top of the stack. First, we present it with its auxiliary strategies:

```
          checkV = map(j, $n, $s)⇝map(j, $n, $s)
findVAndApply(S) = ITE(checkV, S,
                   One3(findAndApply(S)))
          upSwap = map($v, $n, map(stackH, $n, $s))⇝
                   map(stackH, $n, map($v, $n, $s))
           upAux = Choice(upSwap,
                   Sequence(One3(upAux), upSwap))
           endUp = map(stackH, $n, map($v, $n, $s))⇝
                   map(t, $n, map($v, $n, $s))
              up = Choice(endUp, upAux)
            copy = map(j, $n, $p) ⇝ map(stackElt,
                   $n, map(j, $n, $p))
           readV = Sequence(findVAndApply(copy), up)
```

The first two strategies (`checkV` and `findVAndApply`) are similar to what we already saw for PNs. The `up` strategy is composed of two auxiliary strategies. `upAux` just swaps two subterms. `endUp` finishes the swapping by setting the name of the auxiliary variable to `t`. The `copy` strategy just copies a term as its name implies. Finally, the reading strategy `readV` does exactly what we need, it finds the variable and moves it up the set of terms to create the stack.

To conclude we can add a mapping to the *userActComp* function as follows:

$$\mathit{userActComp} : \mathit{read}_v \mapsto \texttt{readV}$$

The translation of write_v is very similar to that of read_v and we leave it as an exercise to the reader.

D. Model checking and optimization

Our approach is also useful to describe the model checking computation. For example, we can define a strategy that computes the whole state space as:

```
calculateSS = Fixpoint(Union(Identity, T1, ..., Tn))
```

where the T_i are the transition strategies. Our tool requires the user to enter only the transition strategies: the state space computation derives from them.

The last application of our language is the description of optimizations. Our language allows us to describe common Decision Diagrams optimizations like hierarchy [6], anonymization [22] and saturation [23].

Hierarchy is an optimization introduced by SDD which allows to group parts of the DD to better handle hierarchical systems. In our framework subterms are translated to hierarchical Decision Diagrams (see Section V). Thus, to employ this technique in our approach it suffices to create a signature supporting the creation of hierarchical clusters. For example, if a system consists of several processes, variables that belong to a single process are grouped together. For example, using the signature in Table I and adding a cluster operation `c: mapping, cluster ↦ cluster` variables gather as follows: `c(map(P1_v1, 0, map(P1_v2, 0, empty), c(map(P2_v1, 0, map(P2_v2, 0, empty)), endc)`, where `Pn_vn` is the nth variable for the nth process. The point of the hierarchy is that each subterm is treated as a subsystem. As each subterm is translated to a standalone DD this is also the case for our approach.

Anonymization is another optimization for DDs. The idea is to remove variable names in the DD to promote sharing and operation caching. In our approach one way of performing anonymization is to rename the variables in the clusters to have identical clusters. For the example of last paragraph anonymization creates the following term: `c(map(v1, 0, map(v2, 0, empty), c(map(v1, 0, map(v2, 0, empty)), endc)`. This is efficient especially if both processes have similar behavior. In fact, since all operations in the DDs are cached, only one process will be effectively computed. The computation for the other will retrieve the results from the cache. The effect is multiplied even more because each rewrite step in a ΣDD rewrites several terms at the same time.

Finally, we can also describe different variants of saturation using strategies. Saturation is a well-known optimisation technique for DDs. Given the flexibility of strategies, we can even describe different saturation strategies for each cluster. An example of a saturation strategy in our framework is the following:

```
Satn(S) =Sequence(Choice(Onen(Satn(S)), FixPoint(S),
          Fixpoint(S))
```

This strategy goes to the last subterm of the terms and does a fixpoint on it. Then, it goes upwards and at each level applies a fixpoint. Of course other more agressive variants of saturation can be described with our approach.

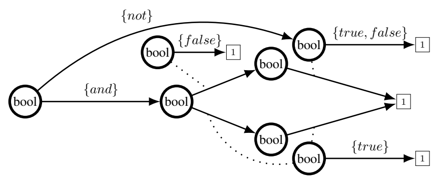

Fig. 6. ΣDD

1) Constraints: Our framework does not currently support non-linear rules, *i.e.,* where a variable appears more than once in the same side of the rule. Indeed, linear rules are directly translatable to DD operations, whereas the translation of non-linear rules can be done in several ways. Depending on the applications, some ways are better than others. Thus, we decided to let the user choose how to implement non-linear rules. For example, the following pattern simulates the `copy` rule, presented earlier, for natural numbers: `map(i, $n, $p) ⇝ map(stackElt, $n, map(i, $n, $p))`:

```
   doubleVar = {suc($n1)⇝ doubleVar(0, 0, suc($n1)),
                0⇝doubleVar(0, 0, 0) }
duplicateVar = doubleVar($n1, $n2, suc($n3))⇝
               doubleVar(suc($n1), suc($n2), $n3)
extractValue = map($v, doubleVar($n1, $n2, 0), $s)⇝
               map(stackH, $n1, map($v, $n2, $s))
        copy = Sequence(
               One2(Sequence(doubleVar,
               Fixpoint(RewriteSet(duplicateVar))),
               extractValue)
```

This principle can be generalized for any sort, if one can define an inductive copy (rules `duplicateVar`) of the values based on the generators (`0,succ` in `duplicateVar`). The strategy `RewriteSet` remembers the number rewritten by `duplicateVar`. It forces the strategy to also keep the terms that were not rewritten. This is one of the few cases where the strategies do not shield the user from the effects of dealing with sets. The pattern needs to be defined only once.

V. Decision Diagrams layer

What we defined until now can be directly implemented in a rewrite term tool but it is not efficient, in particular when there is a lot of similarities in the computed terms. ΣDDs, introduced in [12], generalize Hierarchical Set Decision Diagram (SDD) [6] and allow to encode sets of terms. The ΣDD framework also provides means to perform set rewrite steps using less computational steps than needed if the operation were performed sequentially. We do not explain ΣDDs in depth here but just show their interest to represent sets of terms with efficient sharing and unicity of the representation.

Figure 6 shows a ΣDD encoding the following set: $\{not(true),\ not(false),\ and(false,true),\ and(false,false),\ and(true,true)\}$. The dotted lines on the edges show the hierarchy of subterms (or sets of subterms) which themselves are ΣDDs. Identical structures are memorised once as the $true$ constant term in the figure. ΣDDs can encode all possible sets of terms of a given Many-Sorted

Model	Instance	Marcie time (s)	StrataGEM time (s)	Number of states
FMS	10	0.557	6.411	$2.50 \cdot 10^{9}$
	20	0.953	7.692	$6.02 \cdot 10^{12}$
	50	23.147	29.819	$4.24 \cdot 10^{17}$
	100	time/out	351.027	$2.70 \cdot 10^{21}$
Kanban	10	0.457	5.976	$1.01 \cdot 10^{9}$
	20	0.583	6.716	$8.05 \cdot 10^{11}$
	50	3.728	10.957	$1.04 \cdot 10^{16}$
	100	48.23	51.881	$1.72 \cdot 10^{19}$
SharedMemory	05	0.441	6.214	1863
	10	0.743	8.429	$1.83 \cdot 10^{6}$
	20	14.522	14.73	$4.45 \cdot 10^{11}$
	50	time/out	109.25	$5.87 \cdot 10^{26}$

TABLE II. RUNTIME COMPARISON

Signature in a unique way [12] and all ΣDDs can be decoded to a valid set of terms. Thus we propose to note the encoding as the bijective function $enc : \mathcal{P}(\mathcal{T}_\Sigma) \rightarrow \Sigma\mathbb{DD}$ [12]. Using this encoding function we now redefine the model checking computation of section II in terms of ΣDDs for efficient computation: `enc(Fixpoint)`$(enc(eval_{t_1} \cup ... \cup eval_{t_j} .. \cup Identity))[enc(s_0)] = states$. The enc operation distribute on all definition of strategies (i.e $enc(T \cup T') = enc(T) \cup enc(T')$ or $enc(Choice(S_1, S_2)) = Choice(enc(S_1), enc(S_2)))$ except the basic rewrite rules that are natively implemented and consequently very efficient.

VI. BENCHMARKS

To assess our approach, we have compared performance of our prototype StrataGEM with the tool Marcie [24], a symbolic model checker based on IDD [8]. It was the best tool for the state space track of the Model Checking Contest (MCC) [3] held at Petri Nets Conference 2014.

Tests are run on a Macbook Pro with 16 GB of RAM, and a 2.5 GHz Intel Core i7 processesor. Each test has a wall-clock time limit of 15 minutes. Models, taken from the MCC [3], are Petri nets in PNML format, translated to StrataGEM's transition system (set rewriting) format.

Results are presented in Table II. Three scalable models are considered: a Flexible Manufacturing System (FMS), a classical Kanban system and a mutual-exclusion for shared memory model. Tools are asked to generate their state spaces. Note that the Shared Memory Problem contains 2651 places and 5050 transitions. For large instances of the models, StrataGEM outperforms Marcie, which is pretty clear for Shared Memory and FMS models. StrataGEM's ability to do clustering, saturation and anonymization allows it treat substantially larger models than Marcie. This supremacy is disputed for smaller instances, mainly due to the overhead induced by the Java VM used by StrataGEM.

These results empirically prove that the great generality of our approach does not impact its performance, as our prototype shows to be more than competitive against a tool based on similar techniques. We must note also that our tool is implemented with a small number of line of Scala code (3700 without the parser) due to its very general and generic principles.

VII. RELATED WORK

We first recall the evolution of the level of abstraction of Decision Diagrams. The original symbolic approach [25] encodes the transition relation as a binary relation between preconditions and post-conditions over Boolean state variables, using Binary Decision Diagrams [9].

In 1994, Pastor *et al.* [26] bring symbolic model checking techniques to Petri nets (PNs), a well-known formalism particularly adapted to express concurrency [27]. A higher level formalism entails a bigger effort in the translation of the semantics as Boolean functions. Pastor *et al.* thus focus mainly on safe PNs, a sub-class in which places cannot contain more than one token at once. The marking of a place is thus a Boolean, and the translation to BDDs is straightforward.

Two generalizations of BDDs appear in the late 1990s: Multi-valued Decision Diagrams (MDDs) [28] by Kam *et al.* on the one hand, and Interval Decision Diagrams (IDDs) [29] by Strehl and Thiele on the other hand. They respectively generalize BDDs to an arbitrary domain and to intervals over rational numbers.

MDDs quickly find a niche in the verification community and are adopted by Miner and Ciardo to perform reachability analysis of PNs [30], by exploiting the concept of locality in PNs. Since both PNs and MDDs handle integers directly, this work closes the semantical gap between the model and the verification data structure, rendering the translation from one to the other more manageable. In 2008, a variant of IDDs are also used for the model checking of Petri nets [8], technique examplified by the tool Marcie [24].

In 2001, Ciardo *et al.* introduce *saturation* [23], that exploits locality to efficiently evaluate fixpoints over DDs. Saturation adds an optimization dimension to symbolic model checking, and allows symbolic CTL model-checking.

Ciardo *et al.* describe their optimization directly on the structure, but the MDDs operations used so far did not suffice. In 2002, Couvreur *et al.* attack this problem by creating the Data Decision Diagrams (DDDs) [31], "a specialized version of the Multi-valued Decision Diagrams representing characteristic functions of sets" [31]. They also introduce a new way to describe symbolic operations through *homomorphisms*, which can express various optimizations including saturation. Couvreur and Thierry-Mieg later improved DDD by adding hierarchy to them, giving birth to the Hierarchical Set Decision Diagrams (SDDs) [6]. Hierarchy is a twofold improvement, as it provides a natural framework for hierarchical formalisms, and also improves performance of DD.

Hostettler proposes Σ Decision Diagrams (ΣDDs) [12], a variant of SDDs that represents sets of terms and allows symbolic rewriting thereof. They are used by the tool AlPiNA [7], a model-checker for high-level Algebraic Petri nets [10].

We wish to highlight the qualitative jump introduced by ΣDDs. As the reader might have noticed, one of the limitations of DD approaches lies in the translation of data types from the program to the verification structure. Until ΣDDs, Decision Diagrams are limited by the data types used by the source program. Even though the theory of MDDs allows the usage of an arbitrary data type, ΣDDs are the first to allow the user to *define custom data types at run-time*.

In [21] we present a Petri net symbolic model checker called StrataGEM entirely based on ΣDDs. We explain our framework for the particular case of Petri nets. The present paper describes our framework in full generality. In this paper, we show how using our approach one can describe the data type operations, the DD operations, and the optimizations with only one language.

VIII. Conclusion and Future Work

We have shown in this work how to translate systematically an SOS semantics to symbolic operations expressed through rewrite rules. The generality of the approach is supported by efficient mechanism for the evaluation of high-level constructs that require multiple passes on the symbolic structure. This automated translation mechanism is intended to ease the use of symbolic structures in model-checking through a process transparent to the user. This is an important step towards a generalized and low-cost usage of symbolic data structures for model-checking. Although not demonstrated in the paper, our symbolic operations are able to describe model-checking (such as LTL or CTL model-checking) algorithms, and not only the systems semantics. The benchmarks show that our implementation can outperform state of the art tools, showing that the technique is usable in practice. We are currently developing extension of this framework in order to compute automatically the inverse state computation of a rewrite system by generating a rewrite system. This is the first step for realising a general CTL checker based on the preset states.

References

[1] A. Valmari, "The State Explosion Problem," in *Lectures on Petri Nets I: Basic Models*, ser. Lecture Notes in Computer Science, W. Reisig and G. Rozenberg, Eds. Springer Berlin Heidelberg, 1998, vol. 1491, pp. 429–528.

[2] K. L. McMillan, *Symbolic model checking*. Springer, 1993.

[3] F. Kordon and D. Buchs, "Model Checking Contest Page," http://mcc.lip6.fr/. [Online]. Available: http://mcc.lip6.fr/

[4] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[5] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg, "Towards Distributed Software Model-Checking using Decision Diagrams," in *25th International Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, vol. 8044. Springer Verlag, July 2013, pp. 830–845.

[6] J.-M. Couvreur and Y. Thierry-Mieg, "Hierarchical decision diagrams to exploit model structure," *Formal Techniques for Networked and Distributed Systems-FORTE 2005*, pp. 443–457, 2005.

[7] S. Hostettler, A. Marechal, A. Linard, M. Risoldi, and D. Buchs, "High-Level Petri Net Model Checking with AlPiNA," *Fundamenta Informaticae*, 2011.

[8] A. Tovchigrechko, "Efficient Symbolic Analysis of Bounded Petri Nets Using Interval Decision Diagrams," Ph.D. dissertation, BTU Cottbus, Dep. of CS, October 2008.

[9] S. B. Akers, "Binary Decision Diagrams," *IEEE Trans. Comput.*, vol. 27, no. 6, pp. 509–516, Jun. 1978.

[10] J. Vautherin, "Parallel systems specifications with coloured Petri nets and algebraic specifications," in *Advances in Petri Nets 1987*, ser. Lecture Notes in Computer Science, G. Rozenberg, Ed. Springer Berlin Heidelberg, 1987, vol. 266, pp. 293–308.

[11] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.

[12] E. López Bóbeda, S. Hostettler, A. Marechal, and D. Buchs, "ΣDD built-in formalization," Université de Genève, Tech. Rep., 2012. [Online]. Available: https://smv.unige.ch//technical-reports/pdfs/TR219

[13] J. A. Goguen and J. Meseguer, "Order-sorted algebra i: equational deduction for multiple inheritance, overloading, exceptions and partial operations," *Theor. Comput. Sci.*, vol. 105, pp. 217–273, November 1992.

[14] P. J. Higgins, "Algebras with a scheme of operators," *Mathematische Nachrichten*, vol. 27, pp. 115–132, 1963.

[15] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The maude ltl model checker," 2002.

[16] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, "Tom: Piggybacking rewriting on java," in *Conference on Rewriting Techniques and Applications - RTA'07*, ser. LNCS, vol. 4533. Paris, France: Springer-Verlag, Jun. 2007, pp. 36–47.

[17] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, "Elan: A logical framework based on computational systems," *Electronic Notes in Theoretical Computer Science*, vol. 4, no. 0, pp. 35 – 50, 1996.

[18] K. Buth, "Simulation of SOS definitions with term rewriting systems," in *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, U.K., April 11-13, 1994, Proceedings*, 1994, pp. 150–164.

[19] J. Barnat, L. Brim, and P. Rockai, "Divine 2.0: High-performance model checking," in *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, 2009, pp. 31–32.

[20] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.

[21] E. López Bóbeda, M. Colange, and D. Buchs, "StrataGEM: A Generic Petri Net Verification FrameworK," in *Proceedings of the 35th International Conference on Application and Theory of Petri Nets and Concurrency*, G. Ciardo and E. Kindler, Eds., Jun. 2014.

[22] S. Hong, F. Kordon, E. Paviot-Adet, and S. Evangelista, "Computing a hierarchical static order for decision diagram-based representation from p/t nets," in *Transactions on Petri Nets and Other Models of Concurrency V*, ser. Lecture Notes in Computer Science, K. Jensen, S. Donatelli, and J. Kleijn, Eds. Springer Berlin Heidelberg, 2012, vol. 6900, pp. 121–140.

[23] G. Ciardo, G. Lüttgen, and R. Siminiceanu, "Saturation: An Efficient Iteration Strategy for Symbolic State—Space Generation," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, T. Margaria and W. Yi, Eds. Springer Berlin Heidelberg, 2001, vol. 2031, pp. 328–342.

[24] M. Heiner, C. Rohr, and M. Schwarick, "MARCIE - Model checking And Reachability analysis done effiCIEntly," in *Proc. Petri Nets 2013*, ser. LNCS, J. Colom and J. Desel, Eds., vol. 7927. Springer, June 2013, pp. 389–399.

[25] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang, "Symbolic model checking: 10^{20} States and beyond," *Information and computation*, vol. 98, no. 2, pp. 142–170, 1992.

[26] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia, "Petri Net Analysis Using Boolean Manipulation," in *15th International Conference on Application and Theory of Petri Nets*. Springer-Verlag, 1994, pp. 416–435.

[27] W. Reisig, *Petri Nets: An Introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[28] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Multi-valued Decision Diagrams: Theory and Applications," in *Multiple-Valued Logic*. Springer Berlin Heidelberg, 1998, vol. 4, pp. 9–62.

[29] K. Strehl and L. Thiele, "Interval Diagram Techniques for Symbolic Model Checking of Petri Nets," in *Proceedings Of The Design, Automation And Test In Europe Conference*, 1999, pp. 756–757.

[30] A. S. Miner and G. Ciardo, "Efficient reachability set generation and storage using decision diagrams," in *Proc. 20th Int. Conf. on Applications and Theory of Petri Nets*. Springer-Verlag, 1999, pp. 6–25.

[31] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.-A. Wacrenier, "Data decision diagrams for Petri net analysis," *Application and Theory of Petri Nets 2002*, pp. 129–158, 2002.