Toward efficient state space generation of algebraic Petri nets

BUCHS, Didier, HOSTETTLER, Steve Patrick

Abstract
Algebraic Petri Nets (APN: Petri Nets + Abstract Algebraic Data Types) are powerful tools to model concurrent systems. Because of their high expressive power, allowing end-user to model more complex systems, State Space Explosion is a big issue in APN. Symbolic Model Checking (SMC) and particularly Decision Diagrams (DD) based symbolic model checking is a proven technique to handle the State Space Explosion for simpler formalisms such as P/T Petri Nets. This paper discusses how to use Binary Decision Diagrams' (BDD) evolutions (Data Decision Diagrams, Set Decision Diagrams, $\Sigma$ DD,...) to tackle aforementioned problem in the APN world. The main contributions of this work are the encoding of any APN model using the DD framework and the notion of Algebraic Cluster that tackles the concurrency induced by the token multiplicity. The discussed algorithms have been implemented in a tool that is freely accessible on http://alpina.unige.ch.

Reference
Toward efficient state space generation of algebraic Petri nets

Didier Buchs  
Centre Universitaire D’Informatique  
Université de Genève

Steve Hostettler  
Centre Universitaire D’Informatique  
Université de Genève

Abstract:  
Algebraic Petri Nets (APN: Petri Nets + Abstract Algebraic Data Types) are powerful tools to model concurrent systems. Because of their high expressive power, allowing end-user to model more complex systems, State Space Explosion is a big issue in APN. Symbolic Model Checking (SMC) and particularly Decision Diagrams (DD) based symbolic model checking is a proven technique to handle the State Space Explosion for simpler formalisms such as P/T Petri Nets. This paper discusses how to use Binary Decision Diagrams’ (BDD) evolutions (Data Decision Diagrams, Set Decision Diagrams, ΣDD, …) to tackle aforementioned problem in the APN world. The main contributions of this work are the encoding of any APN model using the DD framework and the notion of Algebraic Cluster that tackles the concurrency induced by the token multiplicity. The discussed algorithms have been implemented in a tool that is freely accessible on http://alpina.unige.ch.
## Contents

1 Introduction 2

2 The modeling formalism: Algebraic Petri Nets 2
   2.1 Algebraic Abstract Data Types ........................................ 3
   2.2 Algebraic Petri Nets .................................................. 4

3 The encoding formalism: Decision Diagrams 6
   Sigma Decision Diagrams ($\Sigma DD$) ..................................... 10

4 Encoding Algebraic Petri Nets 11
   4.1 Encoding the State Space .............................................. 11
   4.2 Encoding the Operational Semantics ............................... 12

5 Managing The Combinatorial Explosion 14
   5.1 Algebraic & Structural Clusters ..................................... 14
      Static analysis ......................................................... 16
      Partial Unfolding ..................................................... 17
   5.2 Hierarchical clustering ................................................. 18
   5.3 Inductive clustering .................................................. 19
   5.4 Clustering composition ............................................... 19
   5.5 Substitutable cluster ................................................ 19

6 Related Work 20

7 Benchmarks 20

8 Conclusion & Future Work 21
KEYWORDS: System design and verification, Higher-level Nets Models, Algebraic Petri Nets, State Space Generation, Decisions Diagrams
1 Introduction

State Space generation is one of the key elements of various existing approaches for Symbolic Model Checking, techniques that are also called model validation [1]. The basic idea of Model Checking is to exhaustively compute the set of all reachable states of the model under validation. Knowing the semantics of the model, one is able to automatically generate the state space, enabling her/him to check properties expressed on the state. This basic concept can be extended for model checking of more advanced notations such as CTL [2] and LTL for which the (labeled) transition system has to be known. Despite its simplicity, this method has a huge drawback induced by some models having intensive use of the concurrency and non-determinism dimensions. Those intrinsic properties usually produce an exponential growth of the state space during exploration of models w.r.t. to the number of concurrent element modeled in the system [3].

Recent techniques based on BDD evolutions [4, 5, 6] limit the growth of the data structure representing symbolically the state space in a logarithmic way. This circumvents the exponential growth of the state space and thus reduces the problems for model checking complex P/T nets. We will call this family of techniques DD (Decision Diagrams) in the rest of the paper.

We propose to generalize these approaches to the Algebraic Petri Nets (APN), which is a formalism that combines Petri nets and Abstract Algebraic Data Types (AADT) [7, 8]. The advantage of algebraic nets is that users are able to describe complex problems in a very condensed way. The user defined data types will be used to control the complexity of the deterministic part of the model, while Petri nets will be used for the control part. Doing model checking on such model is difficult and leads to problems for managing the state space. One technique may be to completely unfold the APN into a P/T net and then to use the same techniques as mentioned before. However, this does not always work due to infinite algebraic domains that must be exhaustively explored if we do not have the help of the dynamic computation of the firing rule.

This paper presents how to encode APN using the DD framework, how to benefit from the algebraic part of the model to provide a partial/total unfolding. We will also discuss how we can extract isolated regions of computation based on what we call Algebraic Clustering and how we can use saturation [9] principles to drastically reduce the number of computations when building the state space.

This article is organized as follow: first we introduce Algebraic nets and the prerequisites required to formally encode the State Space Representation and Generation using the Decision Diagram Framework. After what, we define the concept of Algebraic Cluster and its implementation. Then we compare and analyze our work to other approaches on usual benchmarks. Finally, we conclude and discuss the open issues and future work. For the sake of simplicity, contributions such as $\Sigma DD$, which encode terms as DD-like structures and Multi-Set Decisions Diagrams are not detailed here and are discussed in technical reports [10, 11].

2 The modeling formalism: Algebraic Petri Nets

Algebraic Petri Nets (APN) [7, 8] are basically an extension of P/T Petri Nets in which, instances of User Defined Data Types called Algebraic Abstract Data Types (ADT) replace
black tokens. Although the semantic of the data types is user-defined, this formalism can be compared to Coloured Petri Nets [12]. In this section we will first define the data part of the formalism namely the Abstract Algebraic Data Types and later introduce Algebraic Petri Nets syntax and semantic.

2.1 Algebraic Abstract Data Types

Informally AADT consists in describing domain names called sorts and by defining operators on them. These operators are described syntactically by their names, domains and co-domains; their semantic is constrained by conditional equations. Our theory and implementation are compatible with the powerful extension of AADT called order sorted algebraic data types [13]. Domains are no longer disjoint but follow inclusion relations based on the given ordering. $\text{SORT}$ and $\text{FUNC}$ are disjoint universes of respectively sort and function names.

**Definition 1 (Sort & S-Sorted Set).** Let $S \subseteq \text{SORT}$ be a finite set of sorts. A $S$-sorted set $A$ is a union of a family of sets indexed by $S$ ($A = \bigcup_{s \in S} A_s$), noted as $A = (A_s)_{s \in S}$. Informally a $S$-Sorted Set is a partitioned set in which the partitions are determined by the sorts names.

Example: $S = \{\text{Nat}, \text{Bool}\}$, $A = \{0, 1, 2, 3, \ldots\}_\text{Nat} \cup \{\text{True, False}\}_\text{Bool}$. Next step is to define a Signature, i.e. names of the operations between the sorts.

**Definition 2 (Order Sorted Signature).** Let $\leq \subseteq (S \times S)$ be a partial order. A order sorted signature is a triple $\Sigma = (S, \leq, F)$, where $S \subseteq \text{SORT}$ is a finite set of sorts, $(S, \leq)$ is a partially ordered set of sorts and $F = (F_{w,s})_{w \in S^*, s \in S}$ is a $(S^* \times S)$-sorted set of function names of $\text{FUNC}$. We often denote a function name $f \in F_{s_1, \ldots, s_n}$ by $f : s_1, \ldots, s_n \rightarrow s$ and $f : s$ if $f \in F_{s}$, where $\epsilon$ denotes the empty word. Moreover $\forall f \in F_{w_1, s_1} \cap F_{w_2, s_2}$ and $w_1 \leq w_2$ then $s_1 \leq s_2$.\footnote{Ordering is extended on $S$ to words of equal length of $S^*$ by $s_1, \ldots, s_n$ if $\forall i, s_i \leq s_i'$ with $1 \leq i \leq n$ and to pairs $S^* \times S$ by $(w, s) \leq (w', s')$ if $w \leq w'$ and $s \leq s'$.}

Example: $S = \{\text{Nat}, \text{Bool}, \text{Int}\}$, $\leq = \{\langle\text{Nat, Int}\rangle\}$, $F = \{0 : \rightarrow \text{Nat, true} : \rightarrow \text{Bool, succ} : \text{Nat} \rightarrow \text{Nat, and} : \text{Bool, Bool} \rightarrow \text{Bool, not} : \text{Bool} \rightarrow \text{Bool}\}$

**Definition 3 (Terms of an Order Sorted Signature).** Let $\Sigma = (S, \leq, F)$ be an order-sorted signature and $X$ be a $S$-sorted set of variables. Let also $s, s'$ be sorts of $S$. The set of terms of $\Sigma$ over $X$ is a $S$-sorted set $T_{\Sigma,X}$, where each set $(T_{\Sigma,X})_s$ is inductively defined as follows:

- $x \in (T_{\Sigma,X})_s, \forall x \in X_s$.
- $f \in (T_{\Sigma,X})_s, \forall f : s' \rightarrow s'$ such that $s' \leq s$ and is called a constant.
- For all operations that are not a constant: $f(t_1, \ldots, t_n) \in (T_{\Sigma,X})_s$.
  - $f : s_1, \ldots, s_n \rightarrow s'$ such that $s' \leq s$ and $\forall t_i \in (T_{\Sigma,X})_{s_i}$ with $1 \leq i \leq n$.

We also define $\tau : (T_{\Sigma,X})_s \rightarrow S$, the typing function $s.t \forall t \in (T_{\Sigma,X})_s$, $\tau(t) = s$. $T_{\Sigma,0}$ or simply $T_{\Sigma}$ denotes the set of closed terms (i.e. without variables).

Example: Using example of Def. 2: $T_{\Sigma,X} = \{0, \text{succ}(0), \text{not}(\text{true}), \ldots\}$
**Definition 4 (Order Sorted Σ-Algebra).** Given an order sorted signature Σ = ⟨S, ≤, F⟩. An Order Sorted Σ-Algebra A is a pair (A, F^A), in which A is a S-sorted set of values s.t. A = A_{s_1} ∪ … ∪ A_{s_n} and F^A a family of partial function such that ∀f ∈ F_w,s, w = s_1 … s_n. ∃f ∈ F^A, defined as f^A = A_{s_1} × … × A_{s_n} → A_s. Alg(Σ) denotes the class of Σ-Algebras.

The terms are naturally extended to form a Σ-Algebra. Applications between algebras, preserving the sorting and operations application, are called Σ-Morphisms. Now that we have a semantical representation of the values, we need a morphism from the syntactical world to the semantical one.

**Theorem (Σ-Morphisms of evaluation).** Given an order sorted signature Σ = ⟨S, ≤, F⟩, ∀A ∈ Alg(Σ), ∃µ : TΣ → A, a unique morphism of evaluation written: [[ ]]^A : TΣ → A.

**Definition 5 (Interpretation).** Given an order sorted signature Σ = ⟨S, ≤, F⟩, ∀A ∈ Alg(Σ). Let X be a set of variables: an interpretation is a S-morphism I : X → A. We extend, uniquely, the previous morphism of evaluation µ such that µ : TΣX → A written [[ ]]^A : TΣ → A.

**Definition 6 (Σ-Equation).** Given an order sorted signature Σ = ⟨S, ≤, F⟩ and X be a S-sorted set of variables. The set E of equations is the set of pairs ⟨l, r⟩, denoted l = r with l, r ∈ (TΣX)_{s∈S}.

Example: Using the example of Def. 2, E = {not(not(x)) = x, and(true, x) = x}.

Applying equations E to a Σ-Algebra splits the term algebra into equivalence classes.

**Definition 7 (Ordered Algebraic specification).** An ordered Algebraic Specification Spec = ⟨Σ, X, E⟩. Alg(Spec) ⊆ Alg(Σ) satisfies the axioms of E and is said to be a model of Spec noted MOD(Spec).

The term algebra TΣX ∈ MOD(Spec) quotiented by the equations into equivalence classes is called the Initial Algebra and is noted TΣAlgebra.

Example: Fig. 1 has two examples of algebraic specifications on the right side. The first one defines the philosophers’ algebra and the second one the forks algebra. It gives the valid elements, valid operations and their semantics.

Moreover, we call generators the minimal set of operators of F that can be combined to build any value of the Initial Algebra (under some conditions). The Initial Algebra is said to be finitely generated by the generators. Operationally we use a technique called Term-Rewriting [14] to find the equivalence classes and to evaluate terms.

### 2.2 Algebraic Petri Nets

The following section defines Algebraic Petri Net (APN) [7, 8] and can be skipped by reader familiar with this notion. APN belong to the family of High Level Nets like the Coloured Petri Nets. Instead of using Black Tokens as in the Standard Petri Nets the places may contain instances of AADT.

Concurrent systems often require different copies of the same data. Hence, we must define the notion of multiset (a set with many copies of the same element, also called bags). We will extend all existing sorts with a new multi-set sort and associated operations.
Definition 8 (Multi-Set). A multiset (also called bag) \([E]\) over a set \(E\) (called \(\text{Dom}(E)\)) is a mapping from \(E\) to \(\mathbb{N}\). Given \(\Sigma = \langle S, \leq, F \rangle\), we define new sorts and operations such as:

\[
S' = S \cup \bigcup_{s \in S} [s] \\
\leq' = \leq \cup \bigcup_{s, s' \in S} ([s], [s']) \\
F' = F \cup \bigcup_{s \in S} \left\{ \begin{array}{l}
\epsilon_s : \rightarrow [s], \\
\_s : s \rightarrow [s], \\
\_s : [s], [s] \rightarrow [s], \\
\_s : [s], [s] \rightarrow [s]
\end{array} \right\}
\]

forming the extended signature \([\Sigma] = \langle S', \leq', F' \rangle\). An extended algebra \([A]\) is naturally given to any \(A \in \text{Alg}(\Sigma)\) that is:

\[- [A]_{[s]} = [f] : A \rightarrow \mathbb{N} \] and \(\epsilon_A(a) = 0\), \(\forall a \in A\),

\[- \forall a \in A, [\_A(a)] = f \text{ s.t. } \forall a_1 \in A f(a_1) = 1 \text{ if } a = a_1, 0 \text{ otherwise}\]

\[- f_1 + [A] f_2 = f_3 \text{ s.t. } f_3(a) = f_1(a) + f_2(a), \forall a \in A\]

\[- f_1 - [A] f_2 = f_3 \text{ s.t. } f_3(a) = f_1(a) - f_2(a), \forall a \in A \text{ and } f_2(a) \leq f_1(a), \text{ otherwise } f_3(a) = 0.\]

Example: the multiset \([a, b, a, c]\) over the set \([a, b, c, d]\) of sort \(s\) is represented by \(f(a) = 2, f(b) = f(c) = 1, f(d) = 0\). \(\epsilon_s\) is the empty multiset, \(\_s\) (resp. \(-_s\)) the union (resp. difference) between multiset and \([\_s\], the singleton. \([a, b, a, c]\) is noted \(2a + b + c\).

Definition 9 (Transition). Given \(S\) \(= \langle X, E, \rightharpoonup \rangle\), a set of places \(P\) with a typing function \(\tau : P \rightarrow S\), the behavior of a transition \(t \in T\) is a triple \(\langle \text{In}, \text{Cond}, \text{Out} \rangle = \text{Beh}_t\):

\[- \text{In} = (\text{In}_p)_{p \in P} \text{ is a set of } P\text{-sorted set of multisets of terms, s.t. } \forall p \in P, \text{In}_p \in (T_X)_{\tau(p)} \text{ and } \text{In}_p\text{ is the label of the arc from } p \text{ to } t.\]

\[- \text{Cond} \subseteq T_X \times T_X \text{ is a set of equalities attached to transition name } t \text{ for this axiom, which is valid if all the relations } a = b \text{ of Cond are valid.}\]

\[- \text{Out} = (\text{Out}_p)_{p \in P} \text{ is a set of } P\text{-sorted set of multisets of terms, s.t. } \forall p \in P, \text{Out}_p \in (T_X)_{\tau(p)} \text{ is the label of the arc from transition } t \text{ to place } p.\]

\[- \text{var(Out)} \subseteq \text{var(In)}.\]

Please note that for the sake of simplicity, we have restrained the input arcs annotations to either closed terms or variables. Free composite terms on input arcs, can be converted to a variable on the input arc plus a condition in the transition. \(\text{Beh}\) is the set of all transition behaviors \(\text{Beh}_t, \forall t \in T\).

Example: In the Dining Philosophers example in Fig. 1, the goThink transition is: \(\text{Beh}_{\text{goThink}} = \langle \{[l]_{\text{HasL}}, [r]_{\text{HasR}}\}, \{([f(l), r]), \{[l, r]_{\text{Fork}}, \{\text{phiOf(l)}_{\text{Think}}\}\}.\)

Definition 10 (Algebraic Petri Net Specification). An Algebraic Petri Net Specification is defined as \(\text{APN-SPEC} = \langle S\ rightharpoonup, T, P, \text{Beh}, m_0 \rangle\), where:

\[- S\ rightharpoonup = \langle X, E, \rightharpoonup \rangle \text{ is an ordered algebraic specification, with } \Sigma = \langle S, \leq, F \rangle.\]

\[- T \text{ is the set of transitions.}\]

\[- P \text{ is the set of place names with its typing function } \tau : P \rightarrow S.\]

\[- \text{Beh} \text{ is the behavior of the transitions.}\]

\[- m_0 \text{ the initial marking.}\]

Example: Fig. 1 is an example of such an Algebraic Petri Net Specification.

Definition 11 (Markings). Given an algebra \(A \in \text{Alg}(\Sigma)\), a marking \(m\) is a \(P\)-sorted set of values of \([A]\). It represents the state of the system. The set of all markings is noted \(M^A\) and the marking of the place \(p \in P\) is noted \(m(p)\).
**Definition 12 (Initial markings).** Given an algebra \( A \in \text{Alg}(\Sigma) \), we call initial marking \( m_0 \) a \( P \)-sorted set of terms of \([A]\).

Example: The initial marking of Fig 1 is: \( m_0 = \{[p0, p(p0)]_{\text{Think}}, [f0, f(f0)]_{\text{Fork}}\} \).

**Definition 13 (Transition system).** Given an algebra \( A \), a specification \( \text{APN-SPEC} = \langle \text{Spec}, T, P, \text{Beh}, m_0 \rangle \), a transition system (TS) over \( A \) and \( \text{APN-SPEC} \) is defined by: \( \text{TS}_A(\text{APN-SPEC}) \subseteq M^A \times T \times M^A \). A transition of this TS is noted \( m \xrightarrow{t} m' \) where \( m, m' \in M^A \) and \( t \in T \). A marking \( m' \) is reachable from \( m \) if \( \exists t_1, t_2, \ldots, t_n \in T, m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} m' \) in several steps.

**Definition 14 (Operational Semantics of Algebraic Petri Nets).** Given an \( \text{APN-SPEC} = \langle \text{Spec}, T, P, \text{Beh}, m_0 \rangle \), an algebra \( A \in \text{MOD}(\text{Spec}) \) and a substitution \( \sigma \), we define the firing rule of transition \( t \in T \) where \( \text{Beh}_t = \langle \text{In}, \text{Cond}, \text{Out} \rangle \) as:

\[
\text{Fire}(t, \sigma) \quad \forall p \in P, \begin{array}{c}
\exists p \in P, \text{In}_p^A \subseteq m(p) \land \bigwedge_{(l, r) \in \text{Cond}} [l]_r = r [l]_r \\
m \xrightarrow{t} \bigcup_{p \in P} (m(p) - \text{In}_p^A) + \text{Out}_p^A)
\end{array}
\]

Informally it means that a transition can be fired if there are sufficient tokens in the input places. Moreover, those tokens must satisfy the conditions. If so, the consumed tokens are removed from the input places and the produced tokens are added to the output places. The rule allow us to define the semantics of a given APN \( \text{apn} = \langle \text{Spec}, T, P, \text{Beh}, m_0 \rangle \) for a model \( A \in \text{MOD}(\text{Spec}) \) (noted \( \text{TS}_A(\text{apn}) \)), the set of all sequence of transitions that can be fired from the initial marking \( m_0 \), the reachable states \( \text{Rech}_A(\text{apn}) \) being all the states of \( \text{TS}_A(\text{apn}) \).

# 3 The encoding formalism: Decision Diagrams

Basic definitions will help us to formally define state space representation and operational semantics using the *Decision Diagrams* (DD) \([4, 5, 10, 11]\). Readers familiar with the concept of DD and their recent evolutions can jump directly to Def. 19 that briefly presents MSDD and SDD which are necessary work for this paper but are only summarized here.
Data Decision Diagrams (DDD) and Set Decision Diagrams (SDD) are both evolutions of the well-known Binary Decision Diagrams (BDD)[15]. While BDD is often seen as representing a Boolean function, it can also be seen as a set of sequences of assignments of Boolean values to variables. DDD (resp. SDD) are similar for assignments of any kind of values (resp. sets) of the form \((\text{var}_1 := \text{val}_1), (\text{var}_2 := \text{val}_2), \ldots, (\text{var}_n := \text{val}_n)\). We note \(E\), the set of variable and \(\forall e \in E\), we note \(\text{Dom}(e)\) the set of values that can be taken by the variable \(e\).

0 represents the empty Decision Diagrams, namely a sequence that finishes with 0 does not exist, 1 represents an existing sequence of assignments, and \(T\) represents the undefined sequence. \(T\) is usually obtained whenever operations are performed on incompatible DDD/SDD sequences (cf. Def.16).

Only the DDD definitions are covered in detail. We only give minimal definition for the more advanced classes of DD (SDD, MSDD, \(\Sigma\)DD). The reader should be able to intuitively understand how the more advanced structures work and should refer to [4, 5, 10, 11] for more details.

**Definition 15 (Data Decision Diagrams).** The DDD set \(D\) is the least set:

- \(\{0, 1, T\} \subseteq D\)
- \(\langle e, \alpha \rangle \in D\) with:
  - \(e \in E\) with \(E\) the set of DDD variables.
  - \(\text{Dom}(e)\) represents the domain of the variable \(e \in E\).
  - \(\alpha : \text{Dom}(e) \to D\), s.t \(x \in \text{Dom}(e)\) and \(\{\alpha(x), 0\}\) is finite.

Notation: \(e \xrightarrow{x} d\) denotes the DDD \(\langle e, \alpha \rangle\) with \(\alpha(x) = d\) and \(\forall y \in \text{Dom}(e)\) s.t \(x \neq y\), \(\alpha(y) = 0\).

**Definition 16 (DDD compatibility).** Two DDD are said compatible iff their sequences are compatible. Two sequences \(s = e_1 \xrightarrow{x_1} \ldots 1\) and \(s' = e'_1 \xrightarrow{x'_1} \ldots 1\) are compatible (noted \(s \approx s'\)) iff: \(s = s' = 1\) or \(s = e \xrightarrow{x} d\) \(\land\) \(s' = e' \xrightarrow{x'} d'\) such that \(e = e'\) and \(d \approx d'\) if \(x = x'\).

[Fig. 2: DDD](image)

The DDD with \(E = \{a, b\}\) and \(\bigcup_{e \in E} \text{Dom}(e) = \{2, 3, 5\}\) on the left side represents following union:

\[
\begin{align*}
  &a \xrightarrow{2} b \xrightarrow{2} 1 + a \xrightarrow{2} b \xrightarrow{3} 1 + a \xrightarrow{5} b \xrightarrow{2} 1 + a \xrightarrow{5} b \xrightarrow{3} 1
\end{align*}
\]

This illustrates the sharing among the encoded states. The Cartesian product of the variables’ domains \((\text{Dom}(a) = \{2, 5\}, \text{Dom}(b) = \{2, 3\})\) is encoded in an efficient way.

Since DDD represent sets, we can define the usual set operations on them such as \(\cup_{\text{ddd}}, \cap_{\text{ddd}}, \setminus_{\text{ddd}}, \setminus_{\text{ddd}}\). For a definition of the set operations on DDD see [4]. The concatenation \((\otimes_{\text{ddd}})\) operation is also defined and concats the second operand to every terminal of the first one.

Unlike work on binary decisions diagrams, operators are not limited to those previously defined. Indeed one of the strengths of the DD-like structure is their support of so-called inductive homomorphisms. Namely operations that are inductively defined on the structure of the DDD and that are compatible with the union operator. This compatibility, called
homomorphism, induces a high efficiency of user defined operations. A homomorphism is a mapping \( \phi \) from \( D \) to itself s.t \( \phi(0) = 0 \) and \( \phi(d \cup d') = \phi(d) \cup \phi(d') \), \( \forall d, d' \in D \).

The union (\( \cup \)) and the composition (\( \circ \)) of two homomorphisms are homomorphisms. Since a decision diagram is inductively defined, operations on them can also be inductively defined. This allows the user to give a local definition of the homomorphism, i.e. what it should do with a given pair \( \langle \text{variable}, \text{value} \rangle \).

**Definition 17 (Inductive Homomorphisms on DDD).**

Let \( \{ \phi_{e,x} \mid e \in E, x \in \text{Dom}(e) \} \) be a family of homomorphisms and \( d_1 \) a DDD:

\[
\forall d \in D, \phi(d) = \begin{cases} 
0 & \text{if } d = 0 \\
 d_1 & \text{if } d = 1 \\
 T & \text{if } d = T \\
 \bigcup_{x \in \text{Dom}(e)} \phi_{e,x}(e \xrightarrow{x} \alpha(x)) & \text{if } d = \langle e, \alpha \rangle 
\end{cases}
\]

is an inductive homomorphism. \( d_1 \) represents the returned ddd whenever a homomorphism reaches the terminal node. Please note that induction is not mandatory, namely each and every homomorphism \( \phi_{e,x} \) decides whether to propagate to the sub-graph.

As for the set operations, inductive homomorphism can be evaluated lazily saving both memory and processing time.

Example: Let suppose we want to define a user-defined function \( \phi_{\text{add}, v_1} \) that adds \( v_1 \) to every variable greater than zero and returns 1 when reaching the terminal.

\[
\phi_{\text{add}, v_1}(e \xrightarrow{x} d) = \begin{cases} 
eq v_1 \phi_{\text{add}, v_1}(d), & \text{if } x > 0 \\
eq v_1 \phi_{\text{add}, v_1}(d), & \text{otherwise}
\end{cases} \quad \phi_{\text{add}, v_1}(1) = 1
\]

To handle more complex structures, being able to assign values to variables is not enough. Set Decision Diagrams (SDD) solve that problem by allowing assignments to be sets. Arcs of the SDD represent a set instead of a value.

**Definition 18 (Set Decision Diagrams).** The SDD set \( \mathbb{S} \) is the least set:

- \( \{0, 1, T\} \subseteq \mathbb{S} \)
- \( \forall e \in \mathbb{B}, \text{Dom}(e) \subseteq \mathbb{S} \)
- \( \langle e, \alpha \rangle \in \mathbb{S} \) with:
  - \( e \in \mathbb{B} \) with \( \mathbb{B} \) the set of all SDD variables.
  - \( \alpha : \pi \rightarrow \mathbb{S} \), with \( \pi = \{a_1, \ldots, a_i, \ldots, a_n\} \) a partition of \( \text{Dom}(e) \) s.t \( \forall a_i, a_j \in \pi, \text{ with } i \neq j, \alpha(a_i) \neq \alpha(a_j) \).

As for DDD, \( e \xrightarrow{x} d \) denotes the SDD(e, \( \alpha \)) with \( \alpha(x) = d \). Compatible sequences, concatenation operator (\( \otimes_{\text{sdd}} \)) and set operators (\( \cup_{\text{sdd}}, \cap_{\text{sdd}} \) and \( \setminus_{\text{sdd}} \)) are defined on SDD. For a definition see [5]. One can define SDD homomorphisms that are similar to their DDD equivalent. One can note that since it is possible to embed DDD into SDD, it is also possible to embed DDD homomorphisms into SDD homomorphisms.

BDD, DDD and SDD can be seen as functions that associates a sequence of assignments to a Boolean (Whether or not the sequence exists). However, limiting the co-domain to Boolean can be too restrictive and one may want to associate a sequence to a natural (i.e. Fig. 4). While we originally defined MSDD in order to handle the concept of MultiSet (i.e. Def. 8) in which a value (a term) exists several times. The definition can be generalized to other codomain.
The SDD on the left side represents the Cartesian product of $p_1$ and $p_2$ that is 9 paths or states. SDD $(e_{sdd} = \{p_1, p_2\})$ embeds DDD $(e_{ddd} = \{a, b, c, d\})$:

\[
\begin{align*}
\text{p1:} & \quad 1 \xrightarrow{a} b \xrightarrow{1} 1 \\
& \quad 0 \xrightarrow{a} b \xrightarrow{1} 1 + p_1 \\
& \quad 0 \xrightarrow{a} b \xrightarrow{0} 1 + p_1 \\
\text{p2:} & \quad 1 \xrightarrow{a} b \xrightarrow{1} 1 + p_1 \\
& \quad 0 \xrightarrow{a} b \xrightarrow{0} 1 + p_1 \\
& \quad 0 \xrightarrow{a} b \xrightarrow{0} 1 + p_1 \\
& \quad 0 \xrightarrow{a} b \xrightarrow{1} 1 + p_1 \\
\end{align*}
\]

Again, the power of the SDD lies in the Cartesian product symbolic encoding. Using SDD, thanks to the sets, we end up with a two-dimensional symbolic encoding.

**Definition 19 (Multi-Set Decision Diagrams).** The MSDD set $\mathcal{M}$ is the least set:

- $(\text{leaf}(n)| n \in \mathbb{N}) \cup \{T\} \subseteq \mathcal{M}$
- $(e, \alpha) \in \mathcal{M}$ with:
  - $e \in \mathcal{E}$ with $\mathcal{E}$ the set of all MSDD variables.
  - $\forall e \in \mathcal{E}, \text{Dom}(e) \subseteq \mathcal{S}$
  - $\alpha : \pi \rightarrow \mathcal{M}$, with $\pi = \{a_0, \ldots, a_i, \ldots, a_n\}$ a partition of $\text{Dom}(e)$ s.t $\forall a_i, a_j \in \pi, i \neq j, \alpha(a_i) \neq \alpha(a_j)$.

As for DDD, $e \xrightarrow{\alpha} d$ denotes the MSDD$(e, \alpha)$ with $\alpha(x) = d$. Set operators $(\cup_{\text{msdd}}, \cap_{\text{msdd}}, \setminus_{\text{msdd}})$ are defined on MSDD. For a definition see [11]. The main difference occurs while operating on the terminals, indeed since the terminal are no more limited to $\{0, 1\}$, we extended the behavior of the set operators $\cup$, $\cap$ and $\setminus$ to the terminals. For canonicity reasons, MSDD can embed any DD but no other MSDD. Given two terminal of $\mathcal{M}$ that are $t, t' \in \mathbb{N}$:

- $\text{leaf}(t) \cup \text{leaf}(t') = \text{leaf}(t + t')$
- $\text{leaf}(t) \cap \text{leaf}(t') = \text{leaf}(t)$ if $t \leq t'$ and $\text{leaf}(t')$ otherwise
- $\text{leaf}(t) \setminus \text{leaf}(t') = \text{leaf}(t - t')$ if $t' \leq t$ and 0 otherwise

The MSDD on the left side represents the union of following MSDDS:

\[
\begin{align*}
& b \xrightarrow{8} j \xrightarrow{2} 3 \cup b \xrightarrow{8} j \xrightarrow{1} 3 \cup b \xrightarrow{4} j \xrightarrow{9} 2
\end{align*}
\]

The semantic of terminators is user-defined, it can be cardinalities or probabilities.
Sigma Decision Diagrams ($\Sigma DD$)

$\Sigma DD$ is a data structure, which is SDD-based. It is able to encode large set of terms in a compact way. As for graph term rewriting, subterms are shared which leads to a great memory saving (see Fig. 5). Moreover, we have developed generic homomorphisms that are, given an algebraic specification (cf. Def. 7), automatically able to perform rewriting of a set of terms. This fits particularly well to our DD-based framework and is described in [10]. Let $\Sigma$ be the S-sorted set of $\Sigma DD$ over signature $\Sigma$.

**Definition 20 ($\Sigma DD$ encoding homomorphism).** Let $\Sigma = \langle S, \leq, F \rangle$ be a signature and $X$ be a set of $S$-sorted set of variables. Let also $T$ be a set of terms over $\langle \Sigma, X \rangle$ noted $T_{\Sigma, X}$. The $\Sigma DD$ of $T \subseteq \mathcal{P}(T_{\Sigma, X})$ is defined by the encoding morphism $en_{\Sigma DD} : \mathcal{P}(T_{\Sigma, X}) \rightarrow \Sigma$.

Globally, the encoding is made by labeling nodes of $\Sigma DD$ with sort names. Then, sequentially use operation names and $\Sigma DD$ for the parameters. This encoding follows rules of compatibility, which is essentially structural except the subsort property inherited from ADT.

$s$ and $s'$ are compatible iff $s \leq s'$ or $s' \leq s$.

**Theorem.** The final fundamental result is that under termination and confluence hypothesis of the axiomatization we provide by rewriting $\text{Rew}$ on $\Sigma DD$ values a valid and complete calculus:

$$\forall t, t' \in T_{\Sigma, X} \big\| [t]_{\sigma} \big\|_{\Sigma}^{T_{\Sigma, X} E} \equiv \big\| t' \big\|_{\sigma} \big\|_{\Sigma}^{T_{\Sigma, X} E} \Leftrightarrow \text{Rew}(en_{\Sigma DD}([t])) = \text{Rew}(en_{\Sigma DD}([t']))$$

From an implementation point of view, we can leverage on the canonicity of the representation in order to implement constant time equality between Decision Diagrams and thus implement efficient caching. All set operations or homomorphisms are applied on an inductive structure and thus each processing step can be put in the cache for further use. This is very useful to save computing time and it allows efficient implemention of the fix-point computation (noted $\text{Hom}^*$).

The figure on the left shows two terms with sharing between subterms $+_{N}(s_{N}(0_{N}), \{s_{N}(s_{N}(0_{N})), s_{N}(0_{N})\})$ being rewritten using the following axioms:

- $+(s(x), y) = s(+(x, y))$
- $+(0, x) = x$

Please note that in this example we rewrite two terms $+(s(0), s(s(0)))$ and $+(s(0), s(0))$ in a single rewriting sequence.
4 Encoding Algebraic Petri Nets

This section is devoted to the use of decision diagrams for encoding one APN and computing its state space. The (denotational) semantics of APN was given in term of one particular algebra. When encoding it is clear that only computable algebra can be used. Classically, the algebra defined by the underlying rewriting system. It is, in order to simplify, the initial term algebra. Therefore in the sequel, it means that instead of \( A \) we will use \( T\equiv E \).

4.1 Encoding the State Space

**Place** A place contains tokens and its content evolves, thus it fits well to the concept of DD variables. Therefore, we can see multisets of tokens in places as assignments to variables.

**Token** In an Algebraic Petri Nets, tokens are terms of a signature, in order to represent such objects we use \( \Sigma^{\text{DD}} \). Tokens are encoded in this kind of DD that leverages on the sharing induced by common subterms as shown in Fig. 3.

**Multi-Set** Since multi-set (bag) can contain several time the same element, we use MSDD, which has the ability to encode the cardinality of a token. In fact, this choice has a drawback: we break the one-state per path paradigm.

To encode one state, let say \( m = \{ [2'0, 3's(0)], 2's(s(0))]_{p_1}, [8's(0), 8's(s(0))]_{p_2} \) we use MSDD of Fig. 6, which has two paths but only represent one state. Since we cannot treat multisets of tokens as sets, we cannot use SDD as a container. To preserve canonicity SDD has to perform set operations on arcs in order to detect non-empty intersection between them (see Def. 18).

**Definition 21 (Marking encoding).** Given an Algebraic Specification \( \text{Spec} = \langle \Sigma, X, E \rangle \) with \( \Sigma = \langle S, \leq, OP \rangle \), a set of places \( P \), the initial term algebra \( T\equiv E \in \text{Mod}(\text{Spec}) \) and let \( M^{\text{EE}} \) be a set of markings. The encoding marking function \( \delta_{\text{marking}} \) is defined by:

\[
\delta_{\text{marking}} : M^{T\equiv E} \rightarrow \mathbb{D} \text{ s.t. } \delta_{\text{marking}}(m) = \bigotimes_{p \in P} p \xrightarrow{\delta_{\text{bag}}(m(p))} 1 \quad 3
\]

\[
\delta_{\text{bag}} : T^{\Sigma\equiv E} \rightarrow \mathbb{M} \text{ s.t.}
\]

\[\delta_{\text{bag}}(\epsilon_s) = 0, \quad s \in S \]

\[\delta_{\text{bag}}([t]_s) = s \xrightarrow{\text{en}_{\Sigma^{\text{DD}}}(t)} 1, \quad s \in S \]

\[\delta_{\text{bag}}(ms + ms') = \delta_{\text{bag}}(ms) \cup \delta_{\text{bag}}(ms'), \quad s \in S \]

\[\delta_{\text{bag}}(ms - ms') = \delta_{\text{bag}}(ms) \setminus \delta_{\text{bag}}(ms'), \quad s \in S \]

\[\text{en}_{\Sigma^{\text{DD}}} : \mathcal{P}(T^{\Sigma\equiv X}) \rightarrow \Sigma^{\text{DD}} \Sigma \text{ which encodes a term as a } \Sigma^{\text{DD}}[10].\]

Tokens with identical cardinality shared their sub-terms.

Fig. 6 represents an Algebraic Petri Net on the left and its DD-encoded initial marking representation on the right. It is worth noting that common parts of terms as well as common parts of the states are shared. Thanks to the canonicity of DD-like structure, no redundant information is kept in memory.

\(^3\)Using Def. 11 we note \( m(p) \) the multiset (bag) contained in place \( p \) in marking \( m \).

\(^4\bigotimes\) represents the “big” concatenation.
4.2 Encoding the Operational Semantics

In this chapter we discuss how to implement the operational semantics of a given Algebraic Petri Net \( apn = \langle Spec, P, T, Beh, m_0 \rangle \) using DD homomorphisms. This will be done by computing with the above defined homomorphisms, for the term algebra, a structure \( d \in \mathbb{D} \) representing symbolically Rech\( _T \Sigma \)(apn) i.e.

\[
\bigcup_{m \in \text{Rech}_T \Sigma (apn)} \delta_{\text{marking}}(m) = d = \text{Hom}_{\text{Beh}}^\ast(\delta_{\text{marking}}(m_0))
\]

This is made by applying the encoded transition relation (as an homomorphism) on already computed markings (starting from the initial marking) until reaching a fix point. As we have seen in Def. 14, firing a transition is done in three steps. Firstly remove (\( H^- \)) the tokens from the places and thus create a substitution, then check (Check) if the substitution obtained in 1, fulfils the conditions, if so add (\( H^+ \)) the produced tokens to the output places, otherwise discard the substitution.

The terms on the arcs can be of two types: either closed or free. In the first case, homomorphisms can be statically defined. In the last case, variables or Free Composite Terms (only on output arcs) need to be rewritten once the substitution has been done and thus requires a dynamic homomorphism. In order to get the maximum out of the caching, DD homomorphisms should be as atomic as possible. It is better to build small operators and then to combine them to get the required behavior. Since DD are Directed Acyclic Graph (DAG), we can only scan them in one direction. This is a challenge when using information stored at place \( q \) in place \( p \) if \( p \) is higher (starting from the root) than \( q \) in the DAG.

Let \( H^- \) (resp. \( H^+ \)): \( P \times T_{\Sigma,X} \rightarrow \mathbb{D} \) be the homomorphism, which removes (resp. puts) tokens of DD variables that represent a place in the network. Let Check be the homomorphism that checks an equation on a given substitution (\( t_{x/y} \) substitutes \( v \) to \( x \) in \( t \)). The static version of \( H^- \) homomorphism goes through the graph and for each variable, checks if it corresponds to the place \( p \). If so it also tests whether there are enough tokens in the place, otherwise it discards the branch by returning 0. The dynamic version does pretty much the same, except that it creates the substitutions by adding a DD (\( var, s \)) before the current place. \( H^+ \) and Check will then use those substitutions to check the conditions and put the produced tokens in the places. For performance reasons, in the implementation the homomorphisms handle (remove/put) several tokens at once.
Check goes through the graph and each time it crosses a variable contained, in either its left or right parameter, it substitutes the variable in the carried equations by the value and checks whether both terms are closed. If so, it rewrites them and tests their equality. If the rewritten terms are equal then the condition holds, if not the substitution is discarded.

Please note that Check can also be easily used to check properties on the computed state space. The static $H^*$ scans the DD until it crosses the place in which it should add $n$ times the token $val$.

$H^*$ firstly checks if the current variable represents the place to update or is part of the substitution. In the first case, it checks whether the term representing the token is already closed, if so the tokens are added to the place, if not it puts a hook in the current position by memorizing the current marking and firing the Recall homomorphism. In the second case, it makes the substitution in the $t$ parameter and propagates itself.

Check

<table>
<thead>
<tr>
<th>$H^*$: Static version</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H^*(p, val)(e \xrightarrow{d} d) =$</td>
</tr>
<tr>
<td>if $e = p$, let $v = \tau(p)$ $\xrightarrow{\text{Rew}(en_{2DD}(val))}$ 1</td>
</tr>
<tr>
<td>${ e \xrightarrow{d} d \text{ if } (x \setminus v) \neq \emptyset }$</td>
</tr>
<tr>
<td>${ \emptyset \text{ otherwise} }$</td>
</tr>
<tr>
<td>else $e \xrightarrow{d} H^*(p, val)(d)$</td>
</tr>
<tr>
<td>$H^*(p, val)(1) = T$</td>
</tr>
</tbody>
</table>

Recall simply walk through the graph and substitutes the variable of its $t$ parameter until it is closed, then it creates a DD $\langle \text{place}, \text{mem} \cup t \rangle$ that will be pulled up by the Reloc homomorphism.

<table>
<thead>
<tr>
<th>$H^*$: Dynamic version</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H^*(p, val)(e \xrightarrow{d} d) =$</td>
</tr>
<tr>
<td>if $e = p \lor e \in \text{var}(t)$</td>
</tr>
<tr>
<td>${ e \xrightarrow{d} \text{ if } \text{var}(t) \neq \emptyset }$</td>
</tr>
<tr>
<td>${ \emptyset \text{ otherwise} }$</td>
</tr>
<tr>
<td>else $e \xrightarrow{d} H^*(p, val)(d)$</td>
</tr>
<tr>
<td>$H^*(p, val)(1) = T$</td>
</tr>
</tbody>
</table>

Reloc (not described here) relocates a variable and its value (greatly inspired by the setExpr homomorphism of [16]).

Up swaps a DD with the one given as parameter comes from [4].
Let’s extend the previous definitions of $H^-(\text{resp. } H^+)$ to $H^+ : P \times T_{\Sigma,X} \rightarrow \mathbb{D}$:

- $H^-(p,e,t) = \text{Id}$,
- $H^+(p,[t]) = H^+(p,t)$ with $t \in T_{\Sigma,X}$,
- $H^+(p,\text{bag} + \text{bag}') = H^+(p,\text{bag}) \circ H^-(p,\text{bag}')$ with $\text{bag}, \text{bag}' \in T_{\Sigma,X}$.

**Definition 22 (Transition encoding).** Let $\text{Beh}_t = (\text{In}, \text{Cond}, \text{Out})$ be a transition behaviour. We define $H^+_{\text{Beh}_t}, \text{Check}_{\text{Beh}_t}$, and $H^*_t$ by:

$$H^+_{\text{Beh}_t} = \bigcup_{p \in P} H^+(p,\text{In}_p), \quad H^*_{\text{Beh}_t} = \bigcup_{p \in P} H^+(p,\text{Out}_p)$$

$$\text{Check}_{\text{Beh}_t} = \bigcup_{(l,r) \in \text{Cond}} \text{check}((l,r))$$

The homomorphism $\text{Hom}_{\text{Beh}_t}$ applies the behaviour of all transition of $T$ by combining the previous operators: $\text{Hom}_{\text{Beh}_t} = \bigcup_{\text{Beh}_t \in \text{Beh}_t} H^+_{\text{Beh}_t} \circ \text{Check}_{\text{Beh}_t} \circ H^*_{\text{Beh}_t}$ and finally we compute the transitive closure: $\text{Hom}^*_{\text{Beh}_t} = (\text{Hom}_{\text{Beh}_t} \cup \text{Id})^*$

From that the transition $\text{takeL}$ from Fig. 1 can be encoded by: $H^-(\text{WaitL},[l]) \circ H^-(\text{Fork},[l]) \circ \text{Check}((f,[l])) \circ H^+(\text{HasL},[l])$

With the previous composition and the Chaining Loop principle explained in [9] one can build the state space. Informally one has only to apply the homomorphisms representing the transitions on the state space, starting from the initial marking, until it reaches a fixpoint:

$$S = \text{Hom}^*_{\text{Beh}_t}(S \cup \text{Id}).$$

Unfortunately the above homomorphisms are not sufficient to provide an efficient state space generation for heavily concurrent models.

## 5 Managing The Combinatorial Explosion

Concurrency and non-determinism are major causes of state space explosion. Let’s divide the model $M$ of a system in $n$ components such as $M = C_1 \times \ldots \times C_n$ and thus $|M| = |C_1| \times \ldots \times |C_n|$. Hence in the worst case, namely when each and every component is independent and thus we end up with the Cartesian product of the states. Therefore, we need a way to symbolically encode it. This is exactly what we get when concatenating decision diagrams representing set of states as shown in Fig. 3 and the best use of them is when independent components are composed. This is why, to efficiently use decision diagrams, we need to separate the system in components, consider them locally and compose their state space.

### 5.1 Algebraic & Structural Clusters

A Cluster is a group of places that are somehow linked together. For example, it could group all the places of a given process. This has been done for P/T nets in [9]. A positive effect
Section 5  Managing The Combinatorial Explosion 15

is that it enables saturation as explained in [16, 9]. Informally, local transitions (transitions without side effect outside of a cluster) are first saturated and then inter-cluster transitions are saturated (for details see [6]). Topological based clustering is called Structural Clustering.

Adding tokens of different kinds (APN) induces another dimension, which may also badly impact the state space explosion. Not only there is different ways to combine the transitions but also a single transition can be fired using different substitutions. We propose to tackle this problem as a generalization of [5]. In some cases, when tokens represent different identities we can come down to the same solution i.e. dispatching the different identities in different clusters. When clustering is based on place and token value, we call it Algebraic Clustering.

Definition 23 (Cluster). C is a finite ordered set of cluster names s.t:

\[ \text{C} = \{c_0, \ldots, c_{n-1}\} \] with \( n > 0 \) the number of cluster and let \( < \) be the order (could be partial) on clusters.

Definition 24 (Clustering Function and places of a Cluster). Given a Spec \( \langle \Sigma, X, E \rangle \), the initial term algebra \( T_{\Sigma,E} \in \text{Mod}(\hat{\text{Spec}}) \) and a set of places \( P \), the clustering \( P \)-family of functions is:

\[ \{ Cl_p : (T_{\Sigma,E})_{r(p)} \rightarrow C : \forall p \in P \} \]

We note \( P_c \subseteq P \) the set of places of cluster \( c \in C \) s.t \( P_c \). Tokens are dispatched among places \( \langle c \rangle \). Let \( c_0 \) be the initial token of a philosopher (resp. fork). In the example of Fig. 1 \( n = 2 \).

- \( Cl_{\text{think}}(p_i) = c_i \)
- \( Cl_{\text{fork}}(f_i) = c_i \) with \( p \in \{\text{wl, forks, hl}\} \)
- \( Cl_{\text{hr}}(f_i) = c_{i+1} \) with \( p \in \{\text{wr, hr}\} \) and \( c_n = c_0 \)

Definition 25 (Clustered Marking). Given a Spec \( \langle \Sigma, X, E \rangle \), a Clustered marking cm is a \( \langle C, P \rangle \)-sorted set of closed terms of \( (T_{\Sigma,E})_{r(p)} \), \( p \in P \). The set of all markings is noted \( MC^{T_{\Sigma,E}} \). We note \( mc(c, p) \) the marking of the place \( p \in P \) in cluster \( c \in C \), such that \( mc : C \times P 
\right) \rightarrow (T_{\Sigma,E})_{r(p)} \). Tokens are dispatched among places and clusters.

Definition 26 (DD Clustered Marking Encoding). Given a Spec \( \langle \Sigma, X, E \rangle \), a set of places \( P \), the initial term algebra \( T_{\Sigma,E} \in \text{Mod}(\hat{\text{Spec}}) \) and a set of clusters \( C \). The clustered marking encoding function \( \delta_{\text{cluster}} \) is defined by :

- \( \delta_{\text{cluster}} : MC^{T_{\Sigma,E}} \rightarrow \Sigma \) s.t. \( \delta_{\text{cluster}}(mc) = \bigotimes_{c \in C} c^\delta_{\text{mark}}(mc(c,p)) \rightarrow 1 \)
- \( \delta_{\text{mark}} \) is identical as in Def.21.
Informally we split the dd which encodes the marking in several sdd, each of them representing a cluster and we concatenate them. The DD variable is named after the cluster. The homomorphisms given in section 4.2 are not foreseen to be applied to clusters, Therefore, we need a kind of homomorphism carrier.

**Definition 27 (Local Application).** Let $c \in C$ be a cluster and $\phi$ be an inductive homomorphism which operates on $D$. 
\[
\text{Local}(c, \phi)(e \xrightarrow{e} d) = \begin{cases} 
\phi(e) \xrightarrow{e} d & \text{if } e = c \\
\text{Local}(c, \phi)(d) & \text{otherwise}
\end{cases}
\]
\[
\text{Local}(c, \phi)(1) = T
\]
Local walk through the SDD until it locates cluster $c$ on which it executes $\phi$.

For the sake of simplicity, we didn’t talk explicitly about the homomorphism local invariance [9]. Informally if a homomorphism has a condition that makes it executable only on particular clusters, it makes it invariant on the other ones. For example Local is invariant on all clusters but $c$. Of course, we used local invariance in our implementation in order to get a linear speedup.

A particular case is when the location of the values is the same for each places, we are in a similar situation of that of [9].

**Definition 28 (Structural cluster).** Given a $\text{Spec} = (\Sigma, X, E)$, the initial term algebra $T_{\Sigma_{\text{eq}}} \in \text{Mod}(\text{Spec})$, a set of places $P$, and $c \in \text{Cluster}$ s.t $\forall a \in (T_{\Sigma_{\text{eq}}})_{r(p)}$ we have $\text{Cl}_p(a) \rightarrow c$, $c$ is called a structural cluster.

**Static analysis**

is possible thanks to the axiomatization of our Data Types, from which we get several benefits. Firstly, we can inspect each transition and see if it is local to a cluster. This locality is then used to saturate the local transitions first and then to propagate the changes as described in [5, 6] in which the technique is called saturation. Because we perform the Static Analysis by recursively applying the generators until we reach a fixpoint it only works on algebra that are finitely generated by their generators.

A transition is local if when it will be fired it consumes and produces tokens in the same unique cluster.

**Definition 29 (Input and output sets of clusters).** Given an APN-SPEC, $T_{\Sigma_{\text{eq}}} \in \text{Mod}(\text{Spec})$ and $\text{beh}_t = \langle \text{In}, \text{Cond}, \text{Out} \rangle$ the behaviour of $t \in T$. The input (resp. output) set of clusters of $t$ under substitution $\sigma$ is defined by:
\[
\text{InCl}_r = \bigcup_{p \in P} \text{Cl}_p(\text{In}_p)^{T_{\Sigma_{\text{eq}}}}, \quad \text{OutCl}_r = \bigcup_{p \in P} \text{Cl}_p(\text{Out}_p)^{T_{\Sigma_{\text{eq}}}}
\]

**Definition 30 (Local Transition).** Given an APN-SPEC, the initial term algebra $T_{\Sigma_{\text{eq}}} \in \text{Mod}(\text{Spec})$, $\text{beh}_t = \langle \text{In}, \text{Cond}, \text{Out} \rangle$ the behaviour of $t \in T$ and $\sigma_t = [\sigma] \models \text{Cond}[\sigma]$. A transition $t$ is local iff:
\[
\forall \sigma \in \sigma_t \Rightarrow \text{InCl}_{r,t} = \text{OutCl}_{r,t} \wedge |\text{InCl}_{r,t}| = 1.
\]

\[\text{We also extends the Cluster function } \text{Cl} \text{ to multisets which returns the set of clusters of their elements, possibly empty.}\]
Informally for any valid substitution (i.e. validating the conditions) the input tokens and the output ones are in the same cluster (local to a cluster).

By applying the cluster function and using Fig. 1, one can prove that \textit{goEat} is local because: \( \forall p \in A_{\text{phil}} \), \( Cl_{\text{think}}(p) = Cl_{\text{wl}}(lF(p)) = Cl_{\text{wr}}(rF(p)) \).

During static analysis, we can also perform a complete unfolding of the APN, if the domains are finite and not too large. Namely, we inspect each and every transition and based on their input/output domains, we create specific static homomorphisms that are very efficient for a given transition with given substitution. The drawback is obviously that the amount of generated homomorphisms is related to the algebra sizes. Another solution is to bind the analysis. This let the user define the upper bound of an algebra. Of course this must be used with great caution since it can lead to incorrect results if the system should produce token beyond the bound and thus no homomorphisms will then be able to process those tokens.

**Definition 31 (Complete Unfolding).** Given an APN-SPEC and \( \text{beh}_t = \langle \text{In}, \text{Cond}, \text{Out} \rangle \) the behaviour of \( t \in T \), \( \sigma_t = \{ \sigma \mid \sigma \models \text{Cond}\} \). The unfolded version of \( \text{Hom}_{\text{Beh}} \) is defined as:

\[
\begin{align*}
H^-_{\sigma,\text{Beh}_t} &= \bigcup_{c \in \text{InCl}_{\sigma,\text{Beh}_t}} \text{Local}(c, \bigcup_{p \in P_T} H^-(p, \text{In}_p\sigma)) \\
H^+_{\sigma,\text{Beh}_t} &= \bigcup_{c \in \text{OutCl}_{\sigma,\text{Beh}_t}} \text{Local}(c, \bigcup_{p \in P_T} H^+(p, \text{Out}_p\sigma))
\end{align*}
\]

\[
\text{Hom}_{\text{Beh}} = \bigcup_{\text{Beh}_t \in \text{Beh}} \bigcup_{\sigma \in \sigma_t} H^-_{\sigma,\text{Beh}_t} \circ H^+_{\sigma,\text{Beh}_t}
\]

We leverage on the clustering axiomatization \( \text{Spec}_{\text{cluster}} \) to know precisely from which cluster the token has to be removed and in which cluster the produced token must be put. Since static analysis only works with correct substitutions, we don’t need to check conditions at runtime.

**Partial Unfolding**

To overcome the problem induced by the size of some algebra, one can exclude it (i.e. ADT) from the static analysis. Obviously, static and dynamic will then be mixed. Since we don’t know in which cluster tokens of this algebra should be. We decide to put all the non-analyzed token in the default cluster namely the cluster \( c_0 \). Therefore, whenever the system encounters an unanalyzed algebra it seeks to remove the token from the default cluster and to put the produced one on this algebra also in the default cluster.

We successfully used both techniques for the distributed database manager example (see, Section 7). Once the cluster function is defined, one gets an efficient state space generation in a push-button manner. Of course, the difficult part is the cluster ADT definition. We plan to ease this exercise by providing the user a dedicated language to express her/him domain knowledge instead of using ADT directly. For example, user may simply tell the framework that philosophers and forks are unique and that \( \text{fork}_i \) belongs to \( \text{philosopher}_j \).
5.2 Hierarchical clustering

The previous definition provides a simple way to compute clusters and to gather them. We would like to enrich this idea by adding a simple procedure to compute clusters of clusters in a systematic way. We will proceed by using the order between clusters and a dedicated function \textit{WalkOn} for computing clusters from clusters following a path among clusters. This idea is compatible with the basic idea of clustering, which is a simpler case in which the \textit{WalkOn} function is linear.

\textbf{Definition 32 (Walking function).} Given a set of Clusters \( C \). A hierarchy of function is defined: \( \text{WalkOn} : C \rightarrow C \) and \( \text{FirstOf} : \rightarrow C \) with acyclicity.

Within this definition some properties have to be ensured:

\textbf{Definition 33 (Walking function closure).} Given a set of clusters and a hierarchy of function: \( \text{WalkOn} : C \rightarrow C \) and \( \text{FirstOf} : \rightarrow C \) with acyclicity. We define:

- closure: \( \forall c, c' \in C, \text{WalkOn}^* (c') = c \Leftrightarrow (\text{WalkOn}(c') = c'') \land (\text{WalkOn}^*(c'') = c) \) and \( \text{WalkOn}^*(c) = \text{WalkOn}(c) \)
- acyclicity: \( \exists c \in C, \text{WalkOn}^*(c) = c \)
- maxterm: \( \exists c \in C, \forall c' \in C, c \neq c', \text{WalkOn}^*(c') = c \)

This lead to a procedure replacing the linear one given in definition 31.

\[ \text{OneWalkSetOn}(c) = \{ c' \in C | \text{WalkOn}(c') = c \} \]

is a one step computation. The starting point is to begin with \( \text{firstOf} \). By repeating those steps on the set \( sc \) we can iteratively traverse backward the clusters and build forward the state space as in figure 7 (HUM needs better explanation!!).

\[ \text{OneWalkSetOn}(c) = \{ c' \in C | \text{WalkOn}(c') = c \} \]

\[ \text{Fig. 7: Inverse function and walking function} \]

Going from clusters to above clusters is done in the same way as before for the leaf clusters (those \( c \) that do not have sub-contextes \( \exists c', \text{WalkOn}(c') = c \)). For the other, they will essentially gather sub clusters without computing their own state space.
Definition 34 (Hierarchical Complete Unfolding). Given an APN-SPEC and \( \text{beh}_t = \langle \text{In}, \text{Cond}, \text{Out} \rangle \) the behaviour of \( t \in T \), \( \sigma_t = \{ \sigma \mid \text{Cond} \sigma \} \). The unfolded version of \( \text{Hom}_{\text{Beh}} \) is defined as:

\[
H_{\sigma, \text{Beh}, \text{WalkOn}, c}^+ = \bigcup_{c \in \text{InCl}_{\text{Beh}}} \text{Local}(c, \bigcup_{p \in P_{\text{cl}}} H^-(p, \text{In}_p \sigma))
\]

\[
H_{\sigma, \text{Beh}, \text{WalkOn}, c}^- = \bigcup_{c \in \text{OutCl}_{\text{Beh}}} \text{Local}(c, \bigcup_{p \in P_{\text{cl}}} H^+(p, \text{Out}_p \sigma))
\]

\[
H^+_{\sigma, \text{Beh}, \text{WalkOn}} = \bigcup_{c \in \text{InCl}_{\text{Beh}}} \text{Local}(c, \bigcup_{p \in P_{\text{cl}}} H^-(p, \text{In}_p \sigma))
\]

\[
H^-_{\sigma, \text{Beh}, \text{WalkOn}} = \bigcup_{c \in \text{OutCl}_{\text{Beh}}} \text{Local}(c, \bigcup_{p \in P_{\text{cl}}} H^+(p, \text{Out}_p \sigma))
\]

\[
\text{Hom}_{\text{Beh}, \text{WalkOn}} = \bigcup_{c \in \text{Beh}} \bigcup_{\sigma \in \sigma_t} H^+_{\sigma, \text{Beh}, \text{WalkOn}} \circ H^-_{\sigma, \text{Beh}, \text{WalkOn}}
\]

5.3 Inductive clustering

One interesting class of problems are those where events are dynamically extending the structure of the system. A typical example is the philosopher problem with \textit{Join} and \textit{Leave} actions. In this case finite clustering is not very efficient because the mapping from elements to clusters has to gather sets of element of the structure (for instance with a modulo function). We propose a different mechanism where a part of the clustering function is inductively defined and so the mapping from places and values to these clusters. It means that the clusters under consideration will grow depending on a parameter, through adaptation of the current DD technology and the addition of bounds we should provide a more powerful system.

5.4 Clustering composition

Previous definition have to be given once for one algebraic net, we think that building clustering should be made in a progressive way and that for given algebraic types they can be reused from existing one. Our approach is to give a kind of cluster module and explain how to coherently combine them (in a similar way as it is done in the algebraic setting).

- Cluster Structure: \( \langle \text{Spec, P, C, Cl, WalkOn, } > \rangle = CS \) with acyclicity and maxterm.
- Union of cluster structures: \( CS_1 \cup CS_2 = CS \) is a cluster structure?

5.5 Substitutable cluster

As a cluster is a set of state based on a set of place and values, we can find for the same set of places similarities between clusters when they can be simply computed from each other. This needs the notion of computing a cluster state from another cluster state and a \( S \)-sorted collection of functions.

A cluster \( c' \) is substitutable by an other one \( c \) derived by a cluster transformation \( F = \{ f_s : A_s \rightarrow A_s | s \in S \} \) if for all events:

\[
\forall m \in RM_c(m_0), \forall p \in P, m'(p, c') = F(m(p, c)) \text{ and } m' \in RM_{c'}(m_0)
\]
Typically the transformation function can be a not constant generator on mono dimensional linear structure (for instance \textit{succ} for \textit{naturals}). In the philosopher example the transformation from one cluster is given by the application of \( F_{\text{fork}} = f \) and \( F_{\text{philo}} = p \). Nothing is against applying \( F \) more than once.

All concepts of cluster marking needs to be redefined according to this new notion. It is also the case for \textit{Local} homomorphisms. The encoding will become:

\[
\delta_{\text{cluster}} : MC^{T_{\alpha}} \to S \quad \text{s.t.} \quad \delta_{\text{cluster}}(mc) = \bigotimes_{c \in C} \delta_{\text{lazymarking}}(mc(c, p))
\]

\[
\delta_{\text{lazymarking}} : C \times T_{\Sigma}^{[\Sigma_{\alpha}]} \to S \quad \text{s.t.}
\delta_{\text{lazymarking}}(c', m') = \begin{cases} 
\delta_{\text{marking}}(m') & \text{ lazycase(F)} \\
 F \delta_{\text{lazymarking}}(c, m) & c \to c', m' = F(m) \\
 1 & \text{ otherwise}
\end{cases}
\]

\( \delta_{\text{marking}} \) is identical as in Def.21.

Having computed all not reducible cluster markings avoid to compute those linked to them by lazy computation function.

6 Related Work

Several tools have been implemented to check High Level Nets such as Colored Petri Net (CPN-AMI, Helena [17]), however very few are designed for Algebraic Petri Net. To our best knowledge, Maria [18] is the only one, which claims to support Algebraic Petri Net. However, it is not based on user-defined Algebraic Data Types and their axiomatisations but on built-in types. Different techniques have been proposed in order to tackle the state space explosion in High-Level Nets. Such as static reductions applied on the net before the search [17], partial order methods used to reduce the exploration of redundant paths, allowing some states to never be explored [17]. Other authors [19] have explored distributed state space exploration to get a linear speed-up. Although these works leverage on modularization they do not apply this concept to the tokens. Moreover, we do not perform static reductions or marking graph reductions based on symmetries and thus we do not restrict the checks that can be made on the state space [3].

7 Benchmarks

We have performed various benchmarks\(^{6,7}\) on our implementation. Firstly the well-known Dining Philosophers (highly concurrent, few computations), the Distributed Database and finally the Leader Election and the Slotted Ring (less concurrent, more computations).

\(^{6}\)Performed on a Mac Book Pro with 1 Intel Core 2 Duo at 2.5 Ghz and 2GB of RAM.
\(^{7}\)An gray box means no test, a - means it did crash or did not complete after 6h.
TABLE 1: Benchmarks for the usual examples

<table>
<thead>
<tr>
<th>Model</th>
<th>States</th>
<th>No Clust.</th>
<th>Partial Unfold.</th>
<th>Total Unfold.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DJ #</td>
<td>Mem (MB)</td>
<td>Time (s)</td>
</tr>
<tr>
<td>Dining Philosopher</td>
<td>5</td>
<td>1364</td>
<td>116D3</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1860E3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>2.53E9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>1.63E28</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>1.2E188</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Distributed Database</td>
<td>5</td>
<td>401</td>
<td>16064</td>
<td>7.6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>196821</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>7.1E7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>5.84E17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Slotted Ring</td>
<td>5</td>
<td>53856</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8.3E9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>1.46E15</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Leader Election</td>
<td>5</td>
<td>219</td>
<td>22E3</td>
<td>13.6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>31302</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>399E4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>1.72E21</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

We compared our implementation to Maria\(^8\)[18] and Helena [17], both well known in the domain of High Level Petri Net Model Checking. As expected the version without unfolding and thus without clustering performs poorly against Maria and Helena. However, with clustering and unfolding enabled (partial or total) it outperforms the other tools. Results are not as good as those for P/T nets from [9] because of the unfolding cost and because some symmetries cannot be exploited due to token heterogeneity (can be improved using a symbolic symbolic approach). However, user gains in expressivity and simplicity since APN models are more tractable than their P/T equivalents.

For Dining Philosopher, partial unfolding didn’t make any sense since the algebras are quite small. However, for the Distributed Database Manager we used partial unfolding by ignoring naturals and total unfolding with a bound that limiting naturals numbers to the number of databases. The cost of static analysis (included in total time in the table) starts to be prohibitive for 35 databases and thus is the partial unfolded more performant in that case. The models we used, our implementation, as well as the programs for Maria and Helena can be found under [http://alpina.unige.ch](http://alpina.unige.ch).

8 Conclusion & Future Work

In this paper, we have presented how to leverage on the powerful DD-like structures to tackle the state space explosion in the case of High-Level Nets such as Algebraic Petri Nets. We have also introduced the concept of Algebraic Cluster that group tokens together and is a generalization of the clusters of [5].

\(^8\)with parameters –compile tmp -Y -R -Z
We first shown a general approach to generate the state space of systems modeled using APN, we explained why this approach alone was not sufficient and how to use Structural and Algebraic Clusters to master the State Space exponential growth. Moreover, we have shown that under certain circumstances (limited finitely generated algebra), we can perform a static analysis, which helps to greatly improve the performances by targeting the dynamic homomorphisms or even by replacing them by static ones.

We plan to extend our work in the following ways:

- Explore techniques to automatically infer the cluster ADT from domain knowledge of the user such as mentioned before. In particular, perform the static analysis by resolution techniques instead of complete unfolding.
- Explore techniques to distribute the computation among several nodes to get a linear speedup.
- Leverage on the structural cluster to be able to work with even higher formalisms such as COOPN [20] by handling multi level clustering.
- Build a complete Eclipse based suite of tools for model checking (incl. CTL).

AlPiNA (Algebraic Petri Net Analyzer) is a free software available under the conditions of the GNU General Public License at: http://alpina.unige.ch

References


SMV Technical Reports

This report is in the series of SMV technical reports. The series editor is Didier Buchs (Didier.Buchs@unige.ch).

Any views or opinions contained in this report are solely those of the author, and do not necessarily represent those of SMV group, unless specifically stated and the sender is authorized to do so.

You may order copies of the SMV technical reports from the corresponding author or the series editor. Most of the reports can also be found on the web pages of the SMV group (http://smv.unige.ch/).