A framework for the definition of variants of high-level Petri nets

E. Kindler[1] and L. Petrucci[2]

[1] Informatics and Mathematical Modelling
Technical University of Denmark (DTU)
DK-2800 Lyngby, DENMARK
eki@imm.dtu.dk
[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
laure.petrucci@lipn.univ-paris13.fr

Abstract. There exist many different variants of high-level Petri nets. Many differences between these variants, however, do not concern the features of the particular versions of Petri nets, but they concern the data types that can be defined and used in the different variants of high-level nets. One famous example of a restricted version of high-level nets are well-formed nets (which are currently standardised as symmetric nets in ISO/IEC 15909-1), which basically restrict the data types to finite sets with a very limited set of operations on them. Due to these restrictions, there exist some more powerful analysis algorithms for symmetric nets.
During the standardisation of high-level nets and some of their variations, it turned out that defining the legal data types and the operations on them is the most difficult part. In particular, these definitions become lengthy and mix Petri net specific issues with data-type specific issues, which often blocks the view for the really relevant parts. Even worse, supposedly simpler versions of high-level nets often are more difficult to define than high-level nets in general.
This paper introduces the concepts and the mathematical tools to ease the definition of new variants and versions of high-level Petri nets: a *framework* for defining variants of high-level nets. The main ingredient of this framework is the concept of *generators*, which we recently introduced for formalising modular PNML, and the newly introduced concept of *constructs*.

Keywords: High-level Petri nets, variations, well-formed nets, symmetric nets.

1 Introduction

There exist many different variants of high-level Petri nets. Some of them have special concepts or are different in the way they are formalised. However, in many cases, the differences between these versions are not with the actual Petri net features, but they concern the data types that can be defined or are built-in to the specific version of Petri nets. One example are well-formed nets [1], which are currently standardised under the name *symmetric nets* in an addendum to ISO/IEC 15909-1. The main idea of symmetric nets is to restrict the data types to finite sets with a very limited set of operations on them; in turn, these restrictions results in some powerful analysis techniques.

The problem with the many variants of high-level Petri nets is that the formalisations are very different, and the actual differences are blurred by mixing the conceptual differences with the standard definitions on a very low level of abstraction. In this paper, we introduce a mathematical framework for the definition of different versions of high-level Petri nets, which separates these issues and helps defining the supported constructs on an adequate level of abstraction. In the end, it is possible to define a specific version of high-level nets by three parameters, which can be defined independently from the actual definition of high-level nets. The main ingredients of this framework are generators, which have proven to be useful — if not necessary — for defining and using high-level Petri nets from modules [2], and the new concept of constructs, which will be used for characterising the legal constructs in the algebras.

In order to validate this framework, we give the definition of several versions of high-level Petri nets known from the literature.

2 An example

In order to illustrate the concepts of high-level nets and to discuss some of the features in which the various versions of high-level nets differ, we start with an example. The example is taken from [3, 4], which models a distributed algorithm for computing the minimal distance of every node (agent) to some distinguished root nodes in some communication network.

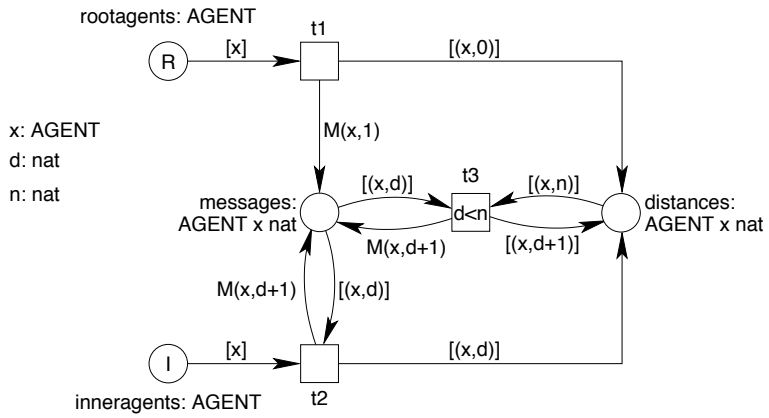

Fig. 1. Example: Minimal distance algorithm

Figure 1 show the algebraic Petri net modelling this algorithm. We assume that there is a *network* of *agents*; the agents could be represented by a set A and the network by a symmetric binary relation $N \subseteq A \times A$. Moreover, there is a distinguished set of *root agents* $R \subseteq A$. Initially, the root agents are represented on place *rootagents*, whereas the other agents, which are called *inner agents*, are represented on place *inneragents*. The algorithm is quite simple: Initially, every root agent sends a message to all its neighbours and stores distance 0 as its own distance; for each neighbour, the message says that it has a distance of at most 1 to some root node. The distance of an agent x is represented by a pair (x, d) on place distances, where x represents the agent, and d its distance. A message to an agent y is represented on the place *messages* by a pairs (y, d), where y represents the agent to which this message is addressed, and d the tentative shortest distance. Sending these initial messages is represented by transition t_1, where $M(x, 1)$ represents a set (resp. a multiset) of all the messages with value 1 to every neighbour of agent x.

Initially, the inner agents cannot do anything. They just wait for a message from one of their neighbours. Once a message arrives for an inner agent x with distance d, it takes that distance d and stores it for itself (represented as pair (x, d) on place *distances*). Moreover, it sends a message to all its neighbours with distance $d+1$, which is represented by $M(x, d+1)$. This behaviour is modelled by transition t_2. But, the inner agents are not finished yet. It might be that later an inner agent receives a distance d that is even shorter than its current distance n. In that case, it takes that shorter distance d as its new distance, and again informs all its neighbours about that. This behaviour is modelled by transition t_3.

Here, we will not discuss the algorithm further. But, we would like to point out some of the notations and concepts used in the algebraic Petri net in Fig. 1. First and foremost, we would like to point out that this is actually not an algebraic net; it is what we call an *algebraic net scheme* [3]. The reason is that the set of agents A, and the way they are connected N, may vary. A can be any set, and N can be any symmetric binary relation over A. This is why the set A is not directly used in the Petri net.

Instead, we use a sort symbol AGENT, which could be interpreted by different sets. Likewise the set of root agents is represented by a constant symbol R and the set of inner agents is represented by the symbol I, and the interpretation of these symbols might be different, depending on which are the root nodes and which are not. Likewise, the structure of the network is represented by the interpretation of the operation symbol M, which produces the messages for all the neighbours.

But, there is another interesting point here. The sort AGENT, the constants R and I, as well as the operation M are specific to this model (actually, they are specific to a class of algorithms, which we call network algorithms). Therefore, their syntax as well as as their legal interpretations need to be defined by the modeller explicitly. In contrast to that, the sort nat does not need to be defined, because it is a standard sort, and also the constants and operations on that sort are standard and have a standard interpretation, which the modeller would not need to define. Actually, the classical approach of algebraic Petri nets [5] forces a modeller to define these sorts and operators even though they are standard. And a modeller would also be forced to explicitly define the pairs for sorts and the boolean operations. In this paper, we will introduce a mechanism that helps avoiding this: Generators help to define the standard sorts and operations of a specific class of high-level nets in a simple and flexible way.

A third observation is that, for the sorts and operations that must be defined by the user, not all possible interpretations are legal. Sometimes, we would like to restrict the interpretations to sorts with a finite set, and also restrict the operations on them. In our example, AGENTS should be finite sets, and the operation M should only be those functions that represent a network (resp. the sending of messages in a network). To this end, this paper introduces the concept of *constructs*. Using this mechanism, we can for example restrict the user defined sorts to the ones that are legal in symmetric nets.

3 Basic Definitions

In this section, we formalise algebraic Petri nets and all the pre-requisites. We introduce the standard concepts of algebraic specifications [6] and of algebraic Petri nets [7–9, 5, 10]. The notation, however, is slightly adjusted for easing the readability of the concepts; the presentation follows the lines of [2] — streamlined a bit for the settings in this paper.

3.1 Basic notations

As usual, $\mathbb{N}$ stands for the set of *natural numbers* (including 0), and $\mathbb{B}$ stands for the set of *booleans*, i.e., $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$. For some set A, A^+ denotes the set of all non-empty finite sequences over A. For some function $f : A \to B$ and some set C, the restriction of f to C is defined as the function $f|_C : A \cap C \to B$ with $f|_C(a) = f(a)$ for all $a \in A \cap C$. For two functions $f : A \to B$ and $g : C \to D$ with disjoint domains A and C, we define $f \cup g$ as the function $(f \cup g) : A \cup C \to B \cup D$ with $(f \cup g)(a) = f(a)$ for all $a \in A$ and $(f \cup g)(c) = g(c)$ for all $c \in C$.

For some set I, a set A together with a mapping $i : A \to I$ is an *I-indexed set* (A, i). The I-indexed set (A, i) is *finite* if A is finite. When i is understood from the context, we often use A for denoting the I-indexed set. For every $j \in I$, we define the set of all elements indexed by j: $A_j = \{a \in A \mid i(a) = j\}$. By definition, all A_j are disjoint. For an I-indexed set (A, i) and some set B, we define $(A, i) \cap B = (A \cap B, i|_B)$.

For some set A, a mapping $m : A \to \mathbb{N}$ is called a *multiset over A* if $\sum_{a \in A} m(a)$ is finite. The set of all multisets over A is denoted by $MS(A)$. For two multisets $m_1, m_2 \in MS(A)$, the operation $+$ is defined pointwise: $m = m_1 + m_2$ is defined by $m(a) = m_1(a) + m_2(a)$ for every $a \in A$. This way, the *addition operation* $+$ is lifted from the natural numbers to multisets. The *empty multiset* is denoted by $[]$ and defined by $[](a) = 0$ for all $a \in A$. This is in line with a special case of our notation that denotes a multiset by enumerating all its elements: $[a_1, a_2, \ldots, a_n]$ (where the number of occurrences of the same element is relevant).

3.2 Signatures and algebras

The idea of high-level nets is that there are different kinds of tokens, which are often called colours. Mathematically, the tokens can come from some set which is associated with a place. Different functions allow for manipulating them. In order to represent these sets and functions, some syntax must be introduced. Here, we use the approach of algebraic nets, where we use signatures for the syntax, and the associated algebras for the meaning.

Definition 1 (Signature). *A* signature $SIG = (S, O)$ *consists of a set of* sort symbols S *(often called sorts for short) and an* S^+*-indexed set of* operation symbols O *such that* S *and* O *are disjoint. The set* $S \cup O$ *is called the set of* symbols *of* SIG*. For some signature* SIG*, we denote the set of its sorts by* S^{SIG} *and the set of its operations by* O^{SIG}*.*

Sometimes, we want to restrict some signature $SIG = (S, (O, i))$ to a subset of symbols A. This is denoted by $SIG|_A$. In the definition, we take care that all the operation symbols operating on an eliminated sort are also eliminated: We define $SIG|_A = (S \cap A, (O', i'))$ where $O' = \{x \in O \mid i(x) \in (S \cap A)^+\}$ and $i' = i|_{O'}$.

Definition 2 (Signature extension). *A signature* SIG' extends a signature SIG *if, for some set* A*,* $SIG'|_A = SIG$*. This is denoted by* $SIG \subseteq SIG'$*. Let* $SIG = (S, O)$ *and* $SIG' = (S', O')$ *be two signatures with a disjoint set of symbols, then we define the union* $SIG \cup SIG' = (S \cup S', O \cup O')$*.*

By definition, $SIG \cup SIG'$ is a signature, which extends both SIG and SIG'.

Definition 3 (Signature homomorphism). *For two signatures* $SIG = (S, O)$ *and* $SIG' = (S', O')$*, a mapping* $\sigma : S \cup O \to S' \cup O'$ *is called a* signature homomorphism*, if for every* $s \in S$ *we have* $\sigma(s) \in S'$ *and for every* $o \in O_{s_1 \ldots s_n}$ *we have* $\sigma(o) \in O'_{\sigma(s_1) \ldots \sigma(s_n)}$*.*

Definition 4 (Algebra). *A* SIG-algebra $\mathcal{A}$ *assigns a* carrier set *to every sort of* SIG *and a* function *to every operation of* SIG*.*

Technically, an algebra $\mathcal{A}$ is a mapping such that, for every $s \in S$, $\mathcal{A}(s)$ is a set and, for every $o \in O_{s_1 \ldots s_n s_{n+1}}$, $\mathcal{A}(o)$ is a function with $\mathcal{A}(o) : \mathcal{A}(s_1) \times \ldots \times \mathcal{A}(s_n) \to \mathcal{A}(s_{n+1})$.

Definition 5 (Algebra extension). *Let* SIG *and* SIG' *be two signatures with* $SIG \subseteq SIG'$*, and let* $\mathcal{A}$ *be a* SIG*-algebra and* $\mathcal{A}'$ *be a* SIG'*-algebra. Algebra* $\mathcal{A}'$ extends *algebra* $\mathcal{A}$*, if* $\mathcal{A}'|_{S^{SIG} \cup O^{SIG}} = \mathcal{A}$*. If* $\mathcal{A}'$ extends $\mathcal{A}$*, we write* $\mathcal{A} \subseteq \mathcal{A}'$*.*

3.3 Variables and terms

The operations of a signature can be used to construct terms, which will be discussed in this section. We start with the definition of variables.

Definition 6 (Variables). *Let* $SIG = (S, O)$ *be a signature. An* S*-indexed set* X *is a set of* SIG-variables*, if* X *is disjoint from* O*.*
$\mathbb{V}^{SIG}$ *denotes the class of all* SIG*-variable sets.*

From the set of operations O of the signature and a set of SIG-variables X, we can construct *terms* of some sort s inductively.

Definition 7 (Terms). *Let* $SIG = (S, O)$ *be a signature and* X *be a set of* SIG-variables*. The set of all* SIG-terms *of sort* s *over a set of variables* X *is denoted by* $\mathbb{T}_s^{SIG}(X)$*. It is inductively defined as follows:*

- $X_s \subseteq \mathbb{T}_s^{SIG}(X)$.
- *For every operation symbol* $o \in O_{s_1 \ldots s_n s_{n+1}}$*, and, for every* k *with* $1 \leq k \leq n$*,* $t_k \in \mathbb{T}_{s_k}^{SIG}(X)$*, we have* $(o, t_1, \ldots, t_n) \in \mathbb{T}_{s_{n+1}}^{SIG}(X)$*.*

When SIG is clear from the context, we also write $\mathbb{T}_s(X)$ instead of $\mathbb{T}_s^{SIG}(X)$. The set of all terms is $\mathbb{T}^{SIG}(X) = \bigcup_{s \in S} \mathbb{T}_s^{SIG}(X)$. Terms without variables are called *ground terms* and are defined by $\mathbb{T}^{SIG} = \mathbb{T}(\emptyset)$ and by $\mathbb{T}_s^{SIG} = \mathbb{T}_s^{SIG}(\emptyset)$.

Sometimes, we need to refer to some terms with variables, but without specifically mentioning the set of variables. The set of such terms is denoted by $\mathbb{T}_s^{SIG}(\mathbb{V}^{SIG})$.

Note that, in practice, terms are often written $o(t_1, \ldots, t_n)$ to make clear that the operation is applied to the arguments. In order to emphasise the syntactical nature of terms, we use the tuple notation $(o, t_1, \ldots, t_n)$ in all our formal definitions.

3.4 Assignment and evaluation

Terms are a purely syntactical construct. In order to give them a meaning, they are evaluated in a given algebra. In order to evaluate terms with variables, we need to bind their variables to some value, which is called a binding or an *assignment*.

Definition 8 (Assignment and evaluation). *Let $SIG = (S, O)$ be a signature, $\mathcal{A}$ a SIG-algebra, and X a set of SIG-variables. An* assignment β of X in $\mathcal{A}$ *is a mapping such that, for every $s \in S$ and every $x \in X_s$, we have $\beta(x) \in \mathcal{A}(s)$.*

An assignment β of variables X in $\mathcal{A}$ can be inductively extended to a mapping $\overline{\beta}$ that applies to all terms $\mathbb{T}^{SIG}(X)$, which is called evaluation of terms in $\mathcal{A}$*:*

- *For every $x \in X$, we define $\overline{\beta}(x) = \beta(x)$.*
- *For every $o \in O_{s_1 \ldots s_{n+1}}$, and every $i \in \{1, \ldots, n\}$ and term $t_i \in \mathbb{T}_{s_i}^{SIG}(X)$, we define $\overline{\beta}((o, t_1, \ldots, t_n)) = \mathcal{A}(o)(\overline{\beta}(t_1), \ldots, \overline{\beta}(t_n))$.*

For an empty set of variables $\emptyset$, there is a unique assignment of $\emptyset$ to $\mathcal{A}$, which we denote with ϵ. The extension $\overline{\epsilon}$ can be used to evaluate ground terms, and is called ground evaluation.

3.5 Generators

In high-level nets and high-level net modules [2] in particular, we often have some sorts provided, and we need to construct other sorts from them in a standard way. For example, we would like to use the product over some existing sorts (see the example in Sect. 2); and, for every sort s, we also need a sort that represents the multiset sort over that sort, $ms(s)$. Moreover, the sets associated with these new sorts are defined based on the sets associated with the underlying sorts. For example, the set associated with $ms(s)$ is the set of all multisets over $\mathcal{A}(s)$, i. e. $MS(\mathcal{A}(s))$.

For that purpose, we need a mechanism for constructing new sorts and operations from some signature and a way to define their meaning. To this end, we introduce *generators*. A generator defines which new sorts and operators can be constructed out of existing sorts, and once the associated sets are known for every sort, what the meaning of the corresponding constructed sorts and operators should be. Since generators are needed anyway, we can also use them for introducing the standard sorts along with their operations (e. g. `nat` or *bool* in our example).

Definition 9 (Generator). *A* generator $G = (GS, GA)$ *consists of*

- *a* signature generator function *GS that, for any given signature $SIG = (S, O)$, returns a signature $GS(SIG)$ such that $SIG \subseteq GS(SIG)$; the signature $GS(SIG)$ is called the signature generated from SIG by the generator G;*
- *an* algebra generator function *GA that, for any SIG-algebra $\mathcal{A}$, returns a $GS(SIG)$-algebra such that the algebra $GA(\mathcal{A})$ extends algebra $\mathcal{A}$.*

In [2], we needed a single generator only, because we were dealing with a single version of Petri nets only. Here, we need different generators for the different versions of high-level nets. Therefore, we will define several generators and operators for constructing new generators out of the existing ones later in the paper. In order to give a feeling for the purpose of a generator, we use the one from [2]

as a first example here; we will introduce other ones later. The basic idea of this example generator $G = (GS, GA)$, is to include, in addition to the existing sorts of some algebra also the booleans, the associated multiset sort $ms(s)$ for every sort s, and all the product sorts. In order to emphasise the syntactical nature, and to distinguish the newly constructed sorts from already existing ones, we use the notation $(bool)$, (ms, s) and $(\times, s_1, \ldots, s_n)$ for these generated sorts. Likewise, the generator will generate the boolean constants $(true)$ and $(false)$ and the standard operations on booleans, the operation $([], s, n)$, which makes a multiset out of n elements, the tupling operation $((), s_1, \ldots, s_n)$, and the projection operation $(pr, i, s_1, \ldots, s_n)$ on the i-th element of a tuple.

Definition 10 (Sort generator). *Let $SIG = (S, O)$ be an arbitrary signature, then $GS(SIG) = (S', O')$ is defined as follows:*

- *S' is the least set for which the following conditions hold:*
  1. *$S \subseteq S'$,*
  2. *$(bool) \in S'$,*
  3. *$(ms, s) \in S'$ for every $s \in S'$, and*
  4. *$(\times, s_1, \ldots, s_n) \in S'$ for all sorts $s_1, \ldots, s_n \in S'$.*
- *O' is the least S'-indexed set for which the following conditions hold:*
  1. *$O \subseteq O'$,*
  2. *$(true), (false) \in O'_{(bool)}$,*
  3. *$(not) \in O'_{(bool)(bool)}$,*
  4. *$(and), (or) \in O'_{(bool)(bool)(bool)}$,*
  5. *$([], s, n) \in O'_{s \ldots s(ms,s)}$ for every sort $s \in S'$ and $n \in \mathbb{N}$, where the number of s elements in the index of O' is n,*
  6. *$(+, s) \in O'_{(ms,s)(ms,s)(ms,s)}$ for every $s \in S'$,*
  7. *$((), s_1, \ldots, s_n) \in O'_{s_1 \ldots s_n(\times, s_1, \ldots, s_n)}$ for all $s_1, \ldots, s_n \in S'$, and*
  8. *for every $0 \leq i \leq n$, $(pr, i, s_1, \ldots, s_n) \in O'_{(\times, s_1, \ldots, s_n)s_i}$.*

Definition 11 (Algebra generator). *Let $\mathcal{A}$ be a SIG-algebra with $SIG = (S, O)$ and let $GS(SIG) = (S', O')$. Then we define $GA(\mathcal{A})$ by:*

- *The mapping of the sorts of $GA(\mathcal{A})$ is defined as follows:*
  1. *$GA(\mathcal{A})|_S = \mathcal{A}|_S$,*
  2. *$GA(\mathcal{A})((bool)) = \mathbb{B}$,*
  3. *$GA(\mathcal{A})((ms, s)) = MS(GA(\mathcal{A})(s))$ for every sort $s \in S'$, and*
  4. *$GA(\mathcal{A})((\times, s_1, \ldots, s_n)) = GA(\mathcal{A})(s_1) \times \ldots \times GA(\mathcal{A})(s_n)$ for all sorts $s_1, \ldots, s_n \in S'$.*
- *The mapping of the operations of $GA(\mathcal{A})$ is defined as follows:*
  1. *$GA(\mathcal{A})|_O = \mathcal{A}|_O$,*
  2. *$GA(\mathcal{A})((true)) = true$ and $GA(\mathcal{A})((false)) = false$,*
  3. *$GA(\mathcal{A})((not)) = \neg$, where $\neg$ is the boolean negation function,*
  4. *$GA(\mathcal{A})((and)) = \wedge$ and $GA(\mathcal{A})((or)) = \vee$, where $\wedge$ and $\vee$ are the boolean conjunction and disjunction functions,*
  5. *$GA(\mathcal{A})(([], s, n))(a_1, \ldots, a_n) = [a_1, \ldots, a_n]$, for every $n \in \mathbb{N}$ and every sort $s \in S'$ and all $a_1, \ldots, a_n \in GA(\mathcal{A})(s)$; i. e. the multiset over s containing exactly the elements $a_1, \ldots, a_n$,*
  6. *$GA(\mathcal{A})((+, s)) = +$ for every sort $s \in S'$, where $+$ denotes the addition of two multisets over $GA(\mathcal{A})(s)$,*
  7. *$GA(\mathcal{A})(((), s_1, \ldots, s_n))(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$ for all $s_1, \ldots, s_n \in S'$ and $a_1 \in GA(\mathcal{A})(s_1), \ldots, a_n \in GA(\mathcal{A})(s_n)$, i. e., the usual tupling, and*
  8. *$GA(\mathcal{A})((pr, i, s_1, \ldots, s_n))((a_1, \ldots, a_n)) = a_i$ for every $0 \leq i \leq n$ and $a_1 \in GA(\mathcal{A})(s_1), \ldots, a_n \in GA(\mathcal{A})(s_n)$; i. e., the usual projection function on the i-th component.*

Note that we need to make sure that all the symbols used in a basic signature *SIG* and introduced by the generators $GS(SIG)$ are interpreted in the same way. In some cases, this might restrict the legal signatures and algebras to which a generator can be applied. In order to avoid overly complex mathematics, we do not introduce an explicit mechanism for that; we rather construct and use generators in a systematic way. For example, in many cases the symbols used in *SIG* are flat and unstructured, whereas the symbols introduced in $GS(SIG)$ are tuples — some of them, like $(bool)$, are 1-tuples. Since this is needed only for making the mathematics work, our examples will use *bool* for $(bool)$ and $ms(s)$ for (ms, s). However, we stick to the technical notations $(bool)$ and (ms, s) in all formal definitions.

Definition 12 (Generator homomorphism). *A signature homomorphism σ from some signature SIG to some signature SIG' carries over to a signature homomorphism σ^G from $GS(SIG)$ to $GS(SIG')$ in a canonical way for any given generator G. In the case of our example, it is defined as follows:*

- *1.* $\sigma^G(s) = \sigma(s)$ *for every* $s \in S$,
  2. $\sigma^G((bool)) = (bool)$,
  3. $\sigma^G((ms, s)) = (ms, \sigma^G(s))$ *for every* $s \in S$, *and*
  4. $\sigma^G((\times, s_1, \ldots, s_n)) = (\times, \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for all* $s_1, \ldots, s_n \in S$.
- *1.* $\sigma^G(o) = \sigma(o)$ *for every operation* $o \in O$,
  2. $\sigma^G((true)) = (true)$ *and* $\sigma^G((false)) = (false)$,
  3. $\sigma^G((not)) = (not)$,
  4. $\sigma^G((and)) = (and)$, *and*
  $\sigma^G((or)) = (or)$,
  5. $\sigma^G(([], s, n)) = ([], \sigma^G(s), n)$ *for every sort* $s \in S$ *and every* $n \in \mathbb{N}$,
  6. $\sigma^G((+, s)) = (+, \sigma^G(s))$ *for every sort* $s \in S$,
  7. $\sigma^G((()), s_1, \ldots, s_n)) = ((), \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for all* $s_1, \ldots, s_n \in S$, *and*
  8. $\sigma^G((pr, i, s_1, \ldots, s_n)) = (pr, i, \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for every* $0 \leq i \leq n$ *and* $s_1, \ldots, s_n \in S$.

3.6 Nets

At last, we introduce the basic notion of *Petri nets*.

Definition 13 (Net). *A* net $N = (P, T, F)$ *consists of two disjoint sets P and T and a set of arcs* $F \subseteq (P \times T) \cup (T \times P)$.

4 Algebraic nets and their behaviour

Now we are prepared to give a first definition of algebraic nets. This definition will be refined later, in order to make it more flexible for defining an algebraic net of a particular kind. In Sect. 4.1, we define algebraic nets; in Sect. 4.2, we define their behaviour. Note that the focus of this paper is not on behaviour; but for completeness sake, we do not want to introduce a formal definition of algebraic nets without a definition of their behaviour.

4.1 Algebraic nets

For a clear separation between syntax and semantics, we distinguish between *algebraic net schemes* and *algebraic nets* [3]. Later we will define different versions of high-level nets, where the generators are one of the main defining factors of a version. For now, we just use the fixed generator G as defined in Sect. 3.5.

By contrast to most classical definitions of algebraic Petri nets and by contrast to our example, we formalise a version of high-level nets, where the scope of a variable is a transition (inspired by [11]). Note that is not a fundamental change; we even have the impression that many people think of variables in Petri nets in this way even when variables are declared globally. But since this more local scope of variables is slightly more complicated to formalise and to use, most formal definitions do not take that view. However, foreseeing some future extensions in the work of ISO/IEC 15909, we deam it necessary to be prepared for this in our formal definition.

Definition 14 (Algebraic net scheme).
An algebraic net scheme *is a tuple* $\Sigma = (N, SIG, sort, vars, l, c, i)$ *consisting of:*

1. *a* net $N = (P, T, F)$,
2. *a* signature SIG,
3. *a* place sort *mapping* $sort : P \to S^{GS(SIG)}$,
4. *a* transition variable *mapping* $vars : T \to \mathbb{V}^{GS(SIG)}$
5. *an* arc label *mapping* $l : F \to \mathbb{T}^{GS(SIG)}(\mathbb{V}^{GS(SIG)})$ *such that:*
   - *for all* $(p,t) \in F \cap (P \times T) : l((p,t)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(vars(t))$
   - *for all* $(t,p) \in F \cap (T \times P) : l((t,p)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(vars(t))$,
6. *a* transition condition *mapping* $c : T \to \mathbb{T}^{GS(SIG)}_{(bool)}(\mathbb{V}^{GS(SIG)})$ *such that* $c(t) \in \mathbb{T}^{GS(SIG)}_{(bool)}(vars(t))$ *for every* $t \in T$,
7. *an* initial marking $i : P \to \mathbb{T}^{GS(SIG)}$ *such that,* $i(p) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}$ *for every place* $p \in P$.

The mapping *sort* assigns a sort to each place, which defines the type of its tokens. The mapping *vars* defines for each transition the set of its variables. The annotations of all arcs attached to a transition may use only these variables; the same applies for the transition condition. Condition 5 formulates this restriction of the variables of the arc annotations as well as the restriction that the arc annotation must denote a multiset over the attached place type. Condition 7 guarantees that the term for the inititial marking denotes a multiset of the respective type.

Definition 15 (Algebraic net). *An* algebraic net *is pair* $(\Sigma, \mathcal{A})$*, where* Σ *is an algebraic net scheme equipped with a* SIG*-algebra* $\mathcal{A}$.

4.2 Behaviour of algebraic nets

For defining the behaviour of an algebraic net, we first need to define markings.

Definition 16 (Marking).
Let $(\Sigma, \mathcal{A})$ *be an algebraic net with* $\Sigma = (N, SIG, sort, vars, l, c, i)$ *and* $N = (P, T, F)$. *A* marking m *of* $(\Sigma, \mathcal{A})$ *is a mapping such that for every place* $p \in P$*, we have* $m(p) \in MS(\mathcal{A}(sort(p)))$.

The operation $+$ *on multisets can be lifted to markings by defining* $m = m_1 + m_2$ *by* $m(p) = m_1(p) + m_2(p)$ *for every place* $p \in P$.

The firing mode of a transition is the set of assignments to its variables such that the transition condition evaluates to true.

Definition 17 (Firing mode).
Let $(\Sigma, \mathcal{A})$ *be an algebraic net with* $\Sigma = (N, SIG, sort, vars, l, c, i)$ *and* $N = (P, T, F)$.

For some transition $t \in T$*, an assignment* β *of the variables* $vars(t)$ *in* $\mathcal{A}$ *is called a* transition mode *of transition* t*, if* $\overline{\beta}(c(t)) = true$.

For any transition t *and any firing mode* β *of* t*, we define the markings* ${}^-t_\beta$ *and* t^+_β *of* $(\Sigma, \mathcal{A})$ *as follows: For every place* $p \in P$

$$ {}^-t_\beta(p) = \begin{cases} \overline{\beta}(l(p,t)) & \text{if } (p,t) \in F \\ [] & \text{otherwise} \end{cases} $$

and

$$ t^+_\beta(p) = \begin{cases} \overline{\beta}(l(t,p)) & \text{if } (t,p) \in F \\ [] & \text{otherwise} \end{cases} $$

Next, we define when and how two markings are reachable from each other by a transition in a firing mode.

Definition 18 (Firing rule).
Let $(\Sigma, \mathcal{A})$ be an algebraic net with $\Sigma = (N, SIG, sort, vars, l, c, i)$ and $N = (P, T, F)$. Moreover, let $t \in T$ be a transition of N, β a firing mode of t, and let m_1 and m_2 be two markings. We say that m_2 is reachable from m_1 by firing t in mode β if there exists a marking m' such that $m_1 = {}^-t_\beta + m'$ and $m_2 = m' + t^+_\beta$. Then, we write $m_1 \xrightarrow{t,\beta} m_2$.

If there exists a sequence $m_1 \xrightarrow{t_1,\beta_1} m_2 \longrightarrow \ldots \longrightarrow m_n \xrightarrow{t_n,\beta_n} m_{n+1}$, we write $m_1 \xrightarrow{*} m_{n+1}$ and say that m_{n+1} is reachable from m_1 *in $(\Sigma, \mathcal{A})$.*

In many publications on Petri nets, $m_1 \xrightarrow{t,\beta} m_2$ would be formalised in a different way by $m_1 \geq {}^-t_\beta$ and $m_2 = (m_1 - {}^-t_\beta) + t^+_\beta$. But, this would require to define the comparision operator $\geq$ on markings and the subtraction operation $-$ first. With our definition, we could avoid that. The only operation necessary for defining the firing rule of Petri nets is the addition operation on markings, $+$.

From these concepts, we can now define the reachability graph as the semantics of an algebraic net.

Definition 19 (Reachability graph).
Let $(\Sigma, \mathcal{A})$ be an algebraic net with $\Sigma = (N, SIG, sort, vars, l, c, i)$ and $N = (P, T, F)$.

We define the initial marking m_0 *of $(\Sigma, \mathcal{A})$ by $m_0(p) = \overline{\epsilon}(i(p))$ for each $p \in P$. We define the set of* reachable markings *of $(\Sigma, \mathcal{A})$ by $\mathcal{M} = \{m \mid m_0 \xrightarrow{*} m\}$. The relation*

$$\mathcal{R} = \{(m_1, (t, \beta), m_2) \mid m_1, m_2 \in \mathcal{M}, t \in T, \text{and } \beta \text{ a mode of } t \text{ with } m_1 \xrightarrow{t,\beta} m_2\}$$

is called the reachability relation *of $(\Sigma, \mathcal{A})$.*

$\Gamma = (\mathcal{M}, m_0, \mathcal{R})$ is called the reachability graph of $(\Sigma, \mathcal{A})$.

5 Generators

As pointed out earlier, different generators can be used for defining different classes of high-level Petri nets. In order to define the respective generators, we define some basic generators and some operations on generators for defining new generators out of existing ones.

5.1 Basic generators

We start with defining some of the basic generators. In these definitions, we assume that $SIG = (S, O)$ is some signature and $\mathcal{A}$ some SIG-algebra.

Identity For technical reasons, we start with introducing the simplest generator of all: the identity generator ID, which does not add any new sorts or operators. This generator is defined by $ID = (ID_{SIG}, ID_{ALG})$, where $ID_{SIG}(SIG) = SIG$ for all signatures SIG. By definition (see Def. 9), ID_{ALG} is then uniquely defined by: $ID_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for all SIG-algebras $\mathcal{A}$ and all symbols x of SIG.

Booleans The generator $BOOL = (BOOL_{SIG}, BOOL_{ALG})$ adds the booleans and their operations to an existing signature: We define $BOOL_{SIG}(SIG) = (S', O')$ by $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(bool)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(true), (false), (not), (and), (or)\}$ with $(true), (false) \in O'_{(bool)}$, $(not) \in O'_{(bool)(bool)}$, and $(and), (or) \in O'_{(bool)(bool)(bool)}$.

We define $BOOL_{ALG}(\mathcal{A})$ by $BOOL_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the symbols $x \in \widehat{S} \cup \widehat{O}$, we define: $BOOL_{ALG}(\mathcal{A})((bool)) = \mathbb{B}$, $BOOL_{ALG}(\mathcal{A})((true)) = true$, $BOOL_{ALG}(\mathcal{A})((false)) = false$, $BOOL_{ALG}(\mathcal{A})((not)) = \neg$, where $\neg$ is the boolean negation function, $BOOL_{ALG}(\mathcal{A})((and)) = \wedge$ and $BOOL_{ALG}(\mathcal{A})((or)) = \vee$, where $\wedge$ and $\vee$ are the boolean conjunction and disjunction functions.

Note that, for some signature SIG, a SIG-algebra $\mathcal{A}$, and some symbol x of SIG, it could happen that $GA(\mathcal{A})(x) \neq \mathcal{A}(x)$. For example, if $(bool)$ occured in SIG but with a completely different meaning. This way, this generator would rule out this algebra as illegal; and we will use this mechanism for enforcing that symbols defined by the generators will be used in this interpretation only.

Restricted booleans Sometimes, we do not even want the full power of booleans. This is for example the case for *Place/Transition nets in high-level net notation* as defined in Part 2 of ISO/IEC 15909. The syntactical definition, however, requires that the trivial transition conditions exist. Therefore, we introduce a generator that introduces a restricted version of the booleans with sort *bool* and only the constant *true*, which can be used in this setting. We thus call this generator $TRUE$. The generator $TRUE = (TRUE_{SIG}, TRUE_{ALG})$ is defined by $TRUE_{SIG}(SIG) = (S', O')$ where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(bool)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(true)\}$ with $(true) \in O'_{(bool)}$. We define $TRUE_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols $x \in \widehat{S} \cup \widehat{O}$, we define: $BOOL_{ALG}(\mathcal{A})((bool)) = \mathbb{B}$ and $BOOL_{ALG}(\mathcal{A})((true)) = true$.

Black tokens For representing Place/Transition nets, we also need a sort that represents the black tokens, which is often called *dots*. The generator DOT adds this sort and a single constant to a signature and the respective algebra. We define the generator $DOT = (DOT_{SIG}, DOT_{ALG})$ by $DOT_{SIG}(SIG) = (S', O')$ where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(dots)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(dot)\}$ with $(dot) \in O'_{(dots)}$. We define $DOT_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $BOOL_{ALG}(\mathcal{A})((dots)) = \{\bullet\}$ and $BOOL_{ALG}(\mathcal{A})((dot)) = \bullet$.

Natural numbers The generator $NAT = (NAT_{SIG}, NAT_{ALG})$ defines the natural numbers. $NAT_{SIG}(SIG) = (S', O')$ is defined by $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(nat)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(0), (succ)\}$ with $(0) \in O'_{(nat)}$, and $(succ) \in O'_{(nat)(nat)}$. We define $NAT_{ALG}(\mathcal{A})$ by $NAT_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $NAT_{ALG}(\mathcal{A})((nat)) = \mathbb{N}$, $NAT_{ALG}(\mathcal{A})((0)) = 0$ and $NAT_{ALG}(\mathcal{A})((succ)) = ++$ where $++$ is the successor operation.

Multisets All versions of high-level Petri nets need some way of denoting multisets. To this end, we define a generator that adds the multisets over the given sorts to the signature and algebra: The generator $MULT = (MULT_{SIG}, MULT_{ALG})$ adds the constructs that are needed for constructing the multisets over the sorts of an algebra. Here, we confine ourselves to a minimal version. Later, we introduce a generator with some more operations on multisets: We define $MULT_{SIG}(SIG) = (S', O')$; where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(ms, s) \mid s \in S\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{([], s, n) \mid s \in S, n \in \mathbb{N}\} \cup \{(+, s) \mid s \in S\}$, where $([], s, n) \in O'_{s \ldots s(ms,s)}$ and $(+, s) \in O'_{(ms,s)(ms,s)}$.

We define $MULT_{ALG}(\mathcal{A})$ by $MULT_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $MULT_{ALG}(\mathcal{A})((ms, s)) = MS(\mathcal{A}(s))$, $MULT_{ALG}(\mathcal{A})(([], s, n))(a_1, \ldots, a_n) = [a_1, \ldots, a_n]$, and $MULT_{ALG}(\mathcal{A})((+, s)) = +$, where $+$ is the addition operation on multisets $MS(\mathcal{A}(s))$.

Extended multisets Some versions of high-level Petri nets have more powerful operations on multisets and for constructing multisets. To this end, we could use a generator that adds these operations. We denote this generator for extended multisets by $MULTX = (MULTX_{SIG}, MULTX_{ALG})$, but we leave the definition open, since it is yet to be decided which operations should be in there. We just assume that $MULT_{SIG}(SIG) \subseteq MULTX_{SIG}(SIG)$ and $MULT_{ALG}(\mathcal{A}) \subseteq MULTX_{ALG}(\mathcal{A})$.

One special extension of the multisets, which we need later for defining symmetric nets with bags is $MULTB$. This generator $MULTB = (MULTB_{SIG}, MULTB_{ALG})$ is defined by $MULTB_{SIG}(SIG) = MULT_{SIG}(SIG) \cup (\emptyset, \{(card), (unique)\})$ where $(card) \in O_{(ms,s)(nat)}$, and $(unique) \in O_{(ms,s)(bool)}$. We define $MULTB_{ALG}(\mathcal{A})$ by $MULTB_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $MULT_{SIG}$. For the other symbols, we define: $MULTB_{ALG}(\mathcal{A})((card))([a_1, \ldots, a_n]) = n$, i. e. the number of elements in the multiset. $MULTB_{ALG}(\mathcal{A})((unique))([a_1, \ldots, a_n])$ is the operator checking that the multiset is actually a set (i. e. there is at most one occurrence of each element).

Products The generator $PROD = (PROD_{SIG}, PROD_{ALG})$ adds all products of the existing sorts and some operations on these products: We define $PROD_{SIG}(SIG) = (S', O')$; where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(\times, s_1, \ldots, s_n) \mid n \in \mathbb{N}, s_1, \ldots, s_n \in S\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{((), s_1, \ldots, s_n) \mid n \in \mathbb{N}, s_1, \ldots, s_n \in$

$S\} \cup \{(pr, i, s_1, \ldots, s_n) \mid n \in \mathbb{N}, i \in \mathbb{N}, 1 \leq i \leq n, s_1, \ldots, s_n \in S\}$ with $((), s_1, \ldots, s_n) \in O'_{s_1 \ldots s_n (\times, s_1, \ldots, s_n)}$ and $(pr, i, s_1, \ldots, s_n) \in O'_{(\times, s_1, \ldots, s_n) s_i}$.

We define $PROD_{ALG}(\mathcal{A})$ by $PROD_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $PROD_{ALG}(\mathcal{A})((\times, s_1, \ldots, s_n)) = \mathcal{A}(s_1) \times \ldots \times \mathcal{A}(s_n)$, $PROD_{ALG}(\mathcal{A})(((), s_1, \ldots, s_n))(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$ for all $s_1, \ldots, s_n \in S$ and $a_1 \in \mathcal{A}(s_1), \ldots, a_n \in \mathcal{A}(s_n)$, and $PROD_{ALG}(\mathcal{A})((pr, i, s_1, \ldots, s_n))(a_1, \ldots, a_n) = a_i$.

Note that *PROD* defines only products of the sorts that exist in the input signature already. It does not define products of products recursively. This recursive construction of products is not necessary here, since we can achieve this by the closure construction on this generator, which will be defined later.

Other sorts and operations Note that there could be many other generators that add some of the commonly used sorts and operators, such as the integers, strings, or even more generic sorts like queues or lists over existing sorts. Since, this paper is more on the framework then on actually defining and classifying all the existing versions of high-level nets, we do not need them here, and therefore do not formally define them here.

5.2 Constructions on generators

From existing basic generators, we can now built more complex generators. The operations needed in this paper are the composition $\circ$, the union $\cup$ and the closure construct *.

Sequential composition The sequential composition of two generators $G_1 = (GS_1, GA_1)$ and $G_2 = (GS_2, GA_2)$ is defined as the function composition, i. e. $G_2 \circ G_1 = (GS_2 \circ GS_1, GA_2 \circ GA_1)$. For some signature *SIG* and some *SIG*-algebra $\mathcal{A}$, this means $(GS_2 \circ GS_1)(SIG) = GS_2(GS_1(SIG))$ and $(GA_2 \circ GA_1)(\mathcal{A}) = GA_2(GA_1(\mathcal{A}))$.

Union Likewise, the union $\cup$ is the union of the two mappings $G_1 \cup G_2 = (GS_1 \cup GS_2, GA_1 \cup GA_2)$. Note that we do not require the two generators to add disjoint symbols; actually, the way generators are defined, both generators will add the original symbols, which is why they are not disjoint. But, we assume (resp. we use and combine generators only in such a way) that they both agree on the meaning of these symbols, i. e. the generated algebras will assign the same meaning to these symbols.

Closure The closure of a generator applies the generator over and over again. Let G, be some generator. We define $G^0 = ID$ and, for each $i \in \mathbb{N}$, we define $G^{i+1} = G \circ G^i$. Then we define $G^* = \bigcup_{i \in \mathbb{N}} G^i$.

For example, we can now define the generator G from Sect. 3.5 from the basic generators above using the following constructions: $G = (MULT \cup PROD)^* \circ BOOL$.

6 Constructs

Some variants of high-level Petri nets differ in the data structures that may be used in the underlying algebras. For example, *symmetric nets* allow for data types with finite domains and a very limited set of operations only. It would be possible to use the concept of generators with some sophisticated and parameterised sort definitions, for explicitly introducing these data types. This however, would introduce much syntactical overhead. Due to this overhead, it would also be very tedious to check whether the generators really do what they are supposed to do, and designing and validating such generators would need much care and experience.

This can be alleviated, if we allow to directly provide the respective algebras in the definition of the algebraic net. This however requires a systematic way to characterise the algebras that are legal for a specific version of high-level Petri nets and those which are not. In this section, we will introduce the concepts for doing so: We slightly extend the concept of signatures to define its syntactical part, along with a set of legal algebras. We call them *constructs*. Then a variant of high-level Petri nets can be defined by a set of such constructs.

6.1 Formal definition of constructs

As mentioned above, a construct is, basically, a signature defining the sorts and operators of the construct. In addition to that, we need to identify the *role* of the sorts and operators, or in general the role of symbols. One role could, for example, be that it is a standard symbol (which is defined via some generator); for example the sort booleans or some of the standard operations on it, which have a fixed meaning in all the algebras and may even occur in different constructs with that same meaning. Another role can be that the symbol must not be used by or occur in other constructs, with the same role, which we call *disjoint*; i. e. these symbols of different constructs do not overlap with other constructs. The precise meaning of these roles will be made clear later in Def. 21. For now, we allow to make these roles explicit.

One of the tricky parts of this definition is that we need to take care that the symbols with a fixed interpretation actually have a fixed interpretation. To this end, we exploit the generators again which define the symbols with a fixed meaning. In this definition of constructs, we need to make sure that a construct is compatible with the used generator.

Definition 20 (Construct).
Let $G = (GS, GA)$ be some generator. A construct *$CON = (SIG, F, D, \mathbb{A})$ consists of a signature SIG, two subsets F and D of symbols of SIG and a class of SIG-algebras $\mathbb{A}$, such that*

1. *$GS(SIG|_{\overline{F}}) \supseteq SIG|_F$ and*
2. *for every $\mathcal{A} \in \mathbb{A}$ we have $GA(\mathcal{A}|_{\overline{F}})|_F = \mathcal{A}|_F$,*

where $\overline{F}$ is the set of all symbols of SIG except the symbols of F. The symbols of F are called the fixed symbols *of CON and the symbols of D are called the* disjoint *symbols of CON.*

The class of all constructs with respect to a generator G is denoted by $\mathcal{C}_G$.

Basically, a construct consists of a signature, and some algebras which give all the legal interpretations. The disjoint symbols D define which symbols may not overlap with other constructs, which will be defined in Def. 21. The fixed symbols F, are the symbols with a fixed meaning. Technically, we require that the fixed symbols in the algebra are the ones which would be added by the generator (condition 1) and also have the same meaning in all algebras as they have in the generated parts (condition 2). Note that we use $\supseteq$, because the generator could introduce many more symbols than the ones occuring in the signature (typically, there are infinitely many symbols in the generator but only finitely many in a construct).

Now, a set of such constructs can be used for characterising some specific algebras as defined below.

Definition 21 (Construct mapping and legal algebras).
Let $G = (GS, GA)$ be some generator and $\mathcal{C}$ be a set of constructs, SIG' be a signature, and $\mathcal{A}'$ be a SIG'-algebra.

A construct mapping *$\mathcal{H}$ for $\mathcal{C}$ and SIG is a set of pairs (CON, h), where $CON \in \mathcal{C}$ is a construct with $CON = (SIG, F, D, \mathbb{A})$ and $h : SIG \to SIG'$ is a signature homormorphism such that the following conditions are met:*

1. *$(h|_{\overline{F}})^G|_F = h|_F$ (i. e. the homomorphism respects the interpretation of the fixed symbols as defined by the generator G).*
2. *For all signatures $SIG'' \subseteq SIG'$ with $SIG' \subseteq GS(SIG'')$, for all $((SIG, F, D, \mathbb{A}), h) \in \mathcal{H}$, and for all symbols x of SIG with $h(x)$ in SIG' but not in SIG'', we have $x \in F$ (i. e. symbols with an interpretation that comes from a generator are a fixed symbol of the construct).*
3. *For all $(CON_1, h_1), (CON_2, h_2) \in \mathcal{H}$ and all symbols $x \in D_1$ and $y \in D_2$ with $h_1(x) = h_2(y)$, we have $CON_1 = CON_2$, $h_1 = h_2$ and $x = y$ (i. e. disjoint symbols of two different constructs do not overlap in SIG').*
4. *For all symbols x' of SIG', there exists a $(CON, h) \in \mathcal{H}$ and a symbol $x \in SIG$ such that $h(x) = x'$ (i. e. all symbols in the signature SIG' are part of at least one construct).*
5. *For every $(CON, h) \in \mathcal{H}$, we have $\mathcal{A}' \circ h \in \mathbb{A}$ (i. e. the part of the algebra corresponding to this construct is one allowed by this construct).*

Then, the algebra $\mathcal{A}'$ is said to be a legally constructed algebra *with respect to $\mathcal{C}$ and $\mathcal{H}$. We also say that $\mathcal{H}$ is a construct mapping from $\mathcal{C}$ to $\mathcal{A}'$.*

The main idea of this definition is that in a legal algebra $\mathcal{A}'$, all parts come from some construct. Condition 1 says that the interpretation of the fixed symbols of a construct is the one defined by the generator if these symbols were not there. Condition 2 makes sure that symbols that have a fixed interpretation according to a generator, have been defined as such in the construct. Condition 3 says that the disjoint symbols of all constructs are mapped to different symbols in SIG', so the constructs do not overlap on these symbols. Condition 4 says that all symbols of SIG' result from one of the constructs, i. e. there are no symbols of SIG' that are not injected by any construct mapping to SIG'. At last, condition 5 says that the semantics of the symbols comes from the construct. Note that in condition 5, h is a signature homomorphism from SIG to SIG'; therefore, $\mathcal{A}' \circ h$ is a SIG-algebra, and exactly this 'part' of SIG must be an algebra allowed by the construct (i. e. an algebra from $\mathbb{A}$).

6.2 Examples of constructs

In this section, we will define some examples of constructs, which will be used for defining symmetric nets and some other versions of high-level nets in Sect. 7.2.

Unordered sets The construct of *unordered sets* UO is defined by $UO = (UOSIG, \emptyset, \{u\}, \mathbb{A}_u)$, where $UOSIG = (\{u\}, \emptyset)$ and $\mathbb{A}_u$ contains every $UOSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(u)$ is a finite set.

This is basically defining a sort symbol u, which in all legal interpretations must be a finite set. Since u is in the set of disjoint symbols of UO, this sort symbol may not overlap with disjoint symbols of the other constructs.

Linearly ordered sets The construct of *linearly ordered sets* LO is defined by $LO = (LOSIG, \{(bool)\}, \{o, lt\}, \mathbb{A}_o)$, where $LOSIG = (\{o, (bool)\}, (\{lt\}, i))$ with $i(lt) = o\,o\,(bool)$ and where $\mathbb{A}_o$ contains every $LOSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(o)$ is a finite set and $\mathcal{A}(lt)$ defines an irreflexive total order on $\mathcal{A}(o)$.

Similarly to unordered sets, linearly order sets define a sort o which is associated with a finite set. In addition, linearly ordered sets define an operation lt which defines a total (irreflexive) order on o. Technically, lt is an operation into the booleans. This is where the fixed symbols come in: $(bool)$ is a fixed symbol of this construct, and therefore must obtain the interpretaion as defined in the generator. Note that this sort $(bool)$ might be used by different constructs. The symbols o and lt are disjoint symbols.

Cyclic sets The construct of *cyclic sets* CS is defined by $CS = (CSSIG, \emptyset, \{c, pred, succ\}, \mathbb{A}_c)$, where $CSSIG = (\{c\}, (\{pred, succ\}, i))$ with $i(pred) = i(succ) = c\,c$ and where $\mathbb{A}_c$ contains every $CSSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(c)$ is a finite set and $\mathcal{A}(succ)$ is a function defining a cycle on all elements of $\mathcal{A}(c)$, and $\mathcal{A}(pred)$ is the inverse of $\mathcal{A}(succ)$.

The construct of cyclic sets is very similar to linearly order sets. The elements are arranged in a cycle, which is expressed by the operators *succ* and *pred*.

Partitions The construct of *partitions* PAR is defined by $PAR = (PARSIG, \emptyset, \{p, f\}, \mathbb{A}_p)$, where $PARSIG = (\{s, p\}, (\{f\}, i))$ with $i(f) = s\,p$ and where $\mathbb{A}_p$ contains every $PARSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(p)$ is a finite set and $\mathcal{A}(f)$ is a surjective function.

Equality The construct of *equality* EQ is defined by $EQ = (EQSIG, \{(bool)\}, \emptyset, \mathbb{A}_e)$, where $EQSIG = (\{e, (bool)\}, (\{eq\}, i))$ with $i(eq) = e\,e\,(bool)$ and where $\mathbb{A}_e$ contains every $EQSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(eq)$ is the equality function on $\mathcal{A}(e)$ (i. e. it evaluates to true if both arguments are the same).

Note that, in this construct, the sort symbol e as well as the operation symbol eq expressing the equality are not in the set of the construct's disjoint symbols. This way, the equality operation may be added to any sort (even to the ones coming from other constructs).

All Likewise, the following construct allows to introduce a constant for a sort s that denotes a multiset in which each element of the set associated with this sort occurs exactly once, which is often denoted by all_s; we use, the notation (all, s) here.

The construct ALL is defined by $ALL = (ALLSIG, \{(ms, s)\}, \emptyset, \mathbb{A}_a)$, where $ALLSIG = (\{s, (ms, s)\}, (\{all\}, i))$ with $i(all) = (ms, s)$ and where $\mathbb{A}_a$ contains every $ALLSIG$-algebra $\mathcal{A}$ with $\mathcal{A}(s) = \{a_1, \ldots, a_n\}$ such that $\mathcal{A}((ms, s)) = MS(\mathcal{A}(s))$ and $\mathcal{A}(all) = [a_1, \ldots, a_n]$.

Simple multisets The construct $MULTSNB$ introduces the construction of a multiset element, the complement of a multiset w.r.t. its carrier set, the difference between two multisets, and for a sort s a constant singleton multiset with (all, s) as its only element. This construct is defined by $MULTSNB = (MULTSNBSIG, \{s, (ms, s)\}, \emptyset, \mathbb{A}_m)$, where $MULTSNBSIG = (\{s, (ms, s), (ms, (ms, s))\}, (\{\mathbf{\{\}}, \tilde{}, \setminus, whole\}, i))$ with $i(\mathbf{\{\}}) = (ms, s)(ms, (ms, s))$, $i(\tilde{}) = (ms, s)(ms, s)$, $i(\setminus) = (ms, s)(ms, s)(ms, s)$, $i(whole) = (ms, (ms, s))$, and where $\mathbb{A}_m$ contains every $MULTSNBSIG$-algebra $\mathcal{A}$ with $\mathcal{A}(\mathbf{\{\}})([b_1, \ldots, b_n]) = [[b_1, \ldots, b_n]]$, $\mathcal{A}(\tilde{})([b_1, \ldots, b_k]) = [c_1, \ldots, c_j]$ such that $[c_1, \ldots, c_j]$ contains exactly one occurrence of all the elements of $\mathcal{A}(s)$ not in $[b_1, \ldots, b_k]$, $\mathcal{A}(\setminus)([b_1, \ldots, b_k], [c_1, \ldots, c_j]) = [b_1, \ldots, b_k] \setminus [c_1, \ldots, c_j]$, i.e. the difference between the two multisets, and finally $\mathcal{A}(whole) = [(all, s)]$.

7 The Framework

Now, we can extend the definition of algebraic nets with respect to the used generator and with respect to some constructs, which will then allow us to define different kinds of algebraic nets. Actually, we use two generators: the first one defines the basic sorts and operations for the tokens on the places. The second generator introduces the necessary multiset structure on top of these basic sorts, which are used to construct the arc annotations and the transition conditions. Note that, in the most general case, the first generator has the full power so that tokens could be multisets and even multisets of multisets, etc. The more interesting versions, however, are the ones with a more restrictive first generator.

7.1 Formal definition

Definition 22 (High-level net version definition). *Let G_1, G_2 be two generators, and let $\mathcal{C}$ be a set of constructs with respect to $G_2 \circ G_1$. Then $K = (G_1, G_2, \mathcal{C})$ is a* version definition of high-level nets.

The main idea of a definition of a version $K = (G_1, G_2, \mathcal{C})$ is that the constructs $\mathcal{C}$ define the signatures and algebras that may be explicitly defined by the user, and the generator G_1 defines which other sorts and operations may be constructed from them. These together define the *basic sorts* of this version of algebraic nets. The generator G_2 defines the multiset sorts and the operations that may be used for the annotations of the net (markings, arc labels and transition conditions).

Definition 23 (Algebraic net of kind K). *Let $K = (G_1, G_2, \mathcal{C})$ be a version definition of high-level nets with $G_1 = (GS_1, GA_1)$ and $G_2 = (GS_2, GA_2)$.*

An algebraic net scheme of kind K is a tuple $\Sigma = (N, SIG, sort, vars, l, c, i)$ with

1. *a* net $N = (P, T, F)$,
2. *a* signature SIG,
3. *a* place sort *mapping* $sort : P \to S^{GS_1(SIG)}$,
4. *a* transition variable *mapping* $vars : T \to \mathbb{V}^{GS_1(SIG)}$,
5. *an* arc label *mapping* $l : F \to \mathbb{T}^{GS_2(GS_1(SIG))}(\mathbb{V}^{GS_1(SIG)})$ *such that:*
   - *for all* $(p, t) \in F \cap (P \times T) : l((p, t)) \in \mathbb{T}^{GS_2(GS_1(SIG))}_{(ms, sort(p))}(vars(t))$
   - *for all* $(t, p) \in F \cap (T \times P) : l((t, p)) \in \mathbb{T}^{GS_2(GS_1(SIG))}_{(ms, sort(p))}(vars(t))$,

6. *a* transition condition *mapping* $c : T \to \mathbb{T}_{(bool)}^{GS_2(GS_1(SIG))}(\mathbb{V}^{GS_1(SIG)})$ *with* $c(t) \in \mathbb{T}_{(bool)}^{GS_2(GS_1(SIG))}(vars(t))$ *for every* $t \in T$, *and*
7. *an* initial marking $i : P \to \mathbb{T}^{GS_2(GS_1(SIG))}$ *such that,* $i(p) \in \mathbb{T}_{(ms,sort(p))}^{GS_2(GS_1(SIG))}$ *for every place* $p \in P$.

An algebraic net scheme Σ *of kind* K *together with a SIG-algebra* $\mathcal{A}$ *and a construct mapping* $\mathcal{H}$ *from* $\mathcal{C}$ *to* $\mathcal{A}$ *form an algebraic net* $(\Sigma, \mathcal{A}, \mathcal{H})$ *of kind* K.

Note that the semantics and the reachability graph for algebraic nets of any kind is the same as the one defined before in Sect. 4.2. The exact details of the generators do not play any role, as long as the booleans and the multiset addition are included.

7.2 Examples of definitions of net versions

At last, we apply this framework for defining some versions of high-level Petri nets. Note that this is mainly meant for demonstrating the use of the framework; a more complete, more systematic and careful definition and classification of most of the relevant high-level Petri net versions is out of the scope of this paper.

P/T-systems We start with the most primitive version of high-level nets, which is Place/Transition systems, just in the setting of high-level nets. In ISO/IEC 15909-2 this version is called *Place/Transition systems in high-level net notation*. The basic idea is that all places are of sort *dots*, which represents the black tokens.

This version can be defined by $PT = (DOT, TRUE \circ MULT, \emptyset)$. Since the set of legal constructs is empty, the signature and algebra provided for an algebraic net of this kind must be empty. Since the first generator DOT only creates the sort *dots* from an empty signature, this is the only legal sort for places. For constructing the legal arc inscriptions, also the sorts and operations generated by $TRUE \circ MULT$ on top of that can be used. As discussed earlier, $TRUE$ is a simple version of the booleans with *true* as the only possible value; this is necessary for technical reasons, since in high-level nets as defined above a transition must have condition.

Symmetric nets Next, we define symmetric nets by the help of a specific generator (products and multisets cannot be built recursively) and the symmetric net constructs. We define the generators for symmetric nets by $SN_1 = PROD \circ BOOL$, $SN_2 = MULT$.

We define the constructs for symmetric nets by $\mathcal{C}_{SN} = \{UO, LO, CS, PAR, EQ, ALL\}$.

Then the definition of the symmetric net version of an algebraic net is $K_{SN} = (SN_1, SN_2, \mathcal{C}_{SN})$.

Symmetric nets with bags Symmetric nets where extended in [12] to allow for manipulating multiset elements in places, on arcs and in transitions conditions. This feature provides additional flexibility for modelling with symmetric nets, without losing the analysis techniques such as the symbolic reachability graph.

The definition of the symmetric net with bags version of algebraic net is $K_{SNB} = (MULT \circ PROD \circ BOOL, MULTB \cup NAT, \mathcal{C}_{SN} \cup \{MULTSNB\})$.

Algebraic Petri nets with fixed arc weight Next, we characterise the version of algebraic nets as defined by Reisig [5]. The major characteristics of this version is that the number of tokens flowing through an arc is always the same, which is why they are sometimes called nets with fixed arc weight (see [13]).

The main point is that the multiset structure of the signature is constructed in a fixed way on top of an arbitrary algebra. This is reflected in the formal definition $AN_1 = (BOOL, MULT, \mathcal{C}_{BOOL})$.

The constructor $BOOL$ is used for using $BOOL$ as a pre-defined sort. All other sorts can be defined by arbitrary constructs over that generator. The multisets only come in via the second constructor $MULT$ (which does not provide any operation for flexible arc weights).

Algebraic Petri nets with flexible arc weight In contrast to that, Kindler and Völzer [13] introduce a version of algebraic nets in which the signature and algebra can define arbitrary operations on multisets. And multiset sorts may even be used as the sort of some places, and multisets of multisets are possible. Therefore, the first generator allows to use this, and the constructs for the algebra may even define some of these operations.

This version can be defined as $AN_2 = (MULT^* \circ BOOL, ID, \mathcal{C}_{MULT^* \circ BOOL})$.

Note that the second generator does not need to contribute any further operations, since all the needed multiset operations are already added by the first generator. Therefore, the second generator is just the identity ID.

General version In principle, algebraic nets with flexible arc weights have all the necessary power. But, to obtain this power, they require that all more powerful operations are defined by the user by providing a signature and an algebra with all the desired operations. It would be much easier, if some of these operations were built-in and could be used without explicitly defining them everytime a new algebraic net is used. This is in particular true for products. Therefore, a more general version would have the products and some more operations on multisets available. Only very special sorts or operations would then come from the signature and algebra defined by the user.

This could look like $AN_3 = ((MULTX \cup PROD)^* \circ (BOOL \cup DOT), ID, \mathcal{C}_{(MULTX \cup PROD)^* \circ (BOOL \cup DOT)})$. Actually, even some other sorts like strings, integers, and list could be included here. But, we leave the exact set of operations and sorts that should be built-in open for a discussion here, as we left the exact definition of $MULTX$ open.

8 Conclusion

In this paper, we have introduced a mathematical framework that allows us to define different versions of high-level Petri nets, with different built-in sorts and operators, and different legal constructs in the underlying algebra. The examples at the end of this paper show that a wide variety of different kinds of high-level net can be defined this way. The main advantage is that the legal constructs and the built-in sorts and operations can be defined independently from the actual definition of algebraic Petri nets. This way, it is much easier to define, to compare and classify different versions of high-level Petri nets and make the built-in sorts and operations and the legal constructs explicit.

Note that for making this possible, we needed to introduce some mathematics, which might be hard to understand for non-experts. But, once this framework is there (and validated by some experts), it can be used in an intuitive way, by selecting and combining the respective constructs and generators. Defining a new version can be done without understanding the details of the underlying framework; one just needs to select the basic building blocks, and combine them with each other, which is demonstrated by the examples in Sect. 7.2.

References

1. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In Jensen, K., Rozenberg, G., eds.: Petri Nets: Theory and Application. Springer-Verlag (1991) 373–396
2. Kindler, E., Petrucci, L.: Towards a standard for modular Petri nets: A formalisation. In Franceschinis, G., Wolf, K., eds.: Application and Theory of Petri Nets 2009, Internat. Conference, Proceedings. Volume 5606 of LNCS., Springer-Verlag (2009) 43–62
3. Kindler, E., Reisig, W.: Algebraic system nets for modelling distributed algorithms. Petri Net Newsletter **51** (1996) 16–31
4. Kindler, E., Völzer, H.: Algebraic nets with flexible arcs. Theoretical Computer Science (2001)
5. Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science **80** (1991) 1–34
6. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1985)

7. Berthomieu, B., Choquet, N., Colin, C., Loyer, B., Martin, J., Mauboussin, A.: Abstract Data Nets combining Petri nets and abstract data types for high level specification of distributed systems. In: Proceedings of VII European Workshop on Application and Theory of Petri Nets. (1986)
8. Vautherin, J.: Parallel systems specifications with coloured Petri nets and algebraic specifications. In Rozenberg, G., ed.: Advances in Petri Nets. Volume 266 of LNCS. Springer-Verlag (1987) 293–308
9. Billington, J.: Many-sorted high-level nets. In: Proceedings of the 3rd International Workshop on Petri Nets and Performance Models, IEEE Computer Society Press (1989) 166–179
10. ISO/IEC: Software and Systems Engineering – High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909 (2004)
11. Schmidt, K.: Verification of siphons and traps for algebraic Petri nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets 1997, Internat. Conference, Proceedings. Volume 1248 of LNCS., Springer-Verlag (1997) 427–446
12. Haddad, S., Kordon, F., Petrucci, L., Pradat-Peyre, J., Trèves, N.: Efficient state-based analysis by introducing bags in Petri nets colour domains. In: Proc. 28th American Control Conference (ACC2009), St Louis, Missouri, USA. (2009) 5018–5025
13. Kindler, E., Völzer, H.: Flexibility in algebraic nets. In Desel, J., Silva, M., eds.: Application and Theory of Petri Nets 1998, 19^{th} International Conference. Volume 1420 of LNCS., Springer-Verlag (1998) 345–364