http://www-lipn.univ-paris13.fr/~petrucci/PAPERS
In Proc. 30th Int. Conf. Application and Theory of Petri Nets (PetriNets'2009), Paris, France, June 2009, volume 5606 of Lecture Notes in Computer Science. Springer, 2009.

Towards a Standard for Modular Petri Nets: A Formalisation

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

[1] Informatics and Mathematical Modelling
Technical University of Denmark
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
petrucci@lipn.univ-paris13.fr

Abstract. When designing complex systems, mechanisms for structuring, composing, and reusing system components are crucial. Today, there are many approaches for equipping Petri nets with such mechanisms. In the context of defining a standard interchange format for Petri nets, modular PNML was defined as a mechanism for modules in Petri nets that is independent from a particular version of Petri nets and that can mimic many composition mechanisms by a simple import and export concept. Due to its generality, the semantics of modular PNML was only informally defined. Moreover, modular PNML did not define which concepts could or should be subject to import and export in high-level Petri nets. In this paper, we formalise a minimal version of modular high-level Petri nets, which is based on the concepts of modular PNML. This shows that modular PNML can be formalised once a specific version of Petri net is fixed. Moreover, we present and discuss some more advanced features of modular Petri nets that could be included in the standard. This way, we provide a formal foundation and a basis for a discussion of features to be included in the upcoming standard of a module concept for Petri nets in general and for high-level nets in particular.
Keywords: Modular Petri Nets, Standardisation, High-Level Nets

1 Introduction

It is well-known that, in order to design large and complex systems, a mechanism to break the system down into smaller pieces is needed. Although Petri nets are often blamed for not having a structuring mechanism, there actually are many proposals for composing Petri nets and for splitting large models into smaller ones (see related work in Sect. 8). Moreover, for industrial size systems, it is not only important to have a system composed of smaller subsystems or modules; a module concept must also cater for re-use and abstraction.

The standard on High-level Petri nets, ISO/IEC 15909 Part 1 [1], however, does not define a concept for modules yet. Structuring issues and net extensions

were left for the future Part 3 of the standard. In this paper, we make a proposal for a module concept for high-level Petri nets and its mathematical underpinning. Note that there are two major directions for constructing nets from some parts, which we call *composition* and *modularity*. In composition, basically, any two or more nets can be composed by one or more composition operators, which gives a new net, which, in principle, can be very different from the original nets. In modularity, we can basically instantiate different *module definitions* with clear *interfaces*; these *instances* can be connected with each other at their interfaces, but their structure is not changed. Here, we propose a modularity approach; still, this approach can mimic the main composition operators for nets like place and transition fusion by the help of *import* and *export nodes* and *symbols* in the interfaces.

The module concept which we propose here is—up to some details—the one from *modular PNML* [2, 3], which was defined along with the early version of the *Petri Net Markup Language*[1] for interchanging all kinds of Petri nets. Actually, modular PNML was more ambitious and more general since it was intended to work for all kinds of Petri nets—in the terminology of PNML, for all *Petri net types*. The downside of modular PNML's generality, however, is that the semantics was defined informally and only in terms of syntactical substitutions, copies, and replacements, which was called flattening. Moreover, a concept for a proper and syntactically correct use of symbols was proposed only very recently [5]. Here, we focus on the module concepts for high-level Petri nets and provide both, some semantical concepts for a more elegant way of dealing with modular high-level Petri nets and a clear semantics.

One problem with most of the mathematical formalisations of high-level Petri nets is that the sort and operation symbols and their underlying meaning (the algebra) is monolithic. This becomes a problem when using and combining symbols from different module instances. In this paper, we solve this problem by a simple concept called *generators*.

2 Introductory example

In this section, we discuss the main concepts of modular PNML with the help of an example. The example as well as its explanation are a revised version of the example from [5], which was based on examples from [2, 3].

2.1 Module definition

Figure 1 shows an example of a module Channel that transmits some information from a place *p1* to a place *p2*. To be more precise, Fig. 1 shows the *module definition*. It consists of two parts: The upper part in the bold-faced box defines the *interface* of the module and its name Channel. The interface consists of

[1] PNML is currently under the final ballot as an interchange format for High-level Petri nets, subject of Part 2 of the standard [4].

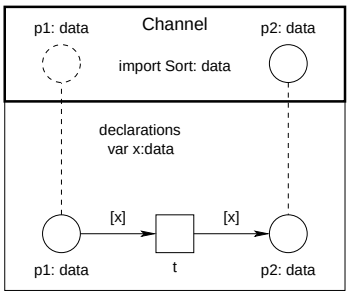


Fig. 1. The module Channel

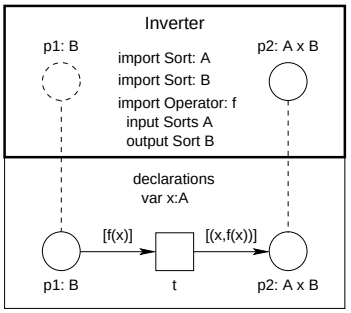


Fig. 2. The module Inverter

different parts: it imports a place (the dashed circle on the left-hand side) and it exports a place (the solid circle on the right-hand side). The difference between import and export nodes will become clear later in this paper. Intuitively, the import place will be provided by the environment of the module when it is used; conversely, the export place is a place that can be used by the environment of the module. In addition to the import and export nodes, the module definition also imports a sort symbol. As indicated by its name, this sort represents the type data that should be transmitted over the channel. In order to make the symbol an import symbol, we use the keyword import[2], the additional text Sort indicates that this symbol is a sort.

The lower part of the module definition in the thinly outlined box is the implementation of the module. Basically, this is a "normal" high-level Petri net. The only difference is that it uses the sort data provided via the interface, and some of its elements refer to the interface. The place on the left-hand side is associated with the import place, while the place on the right-hand side is associated with the export place, as graphically indicated by the two dashed lines.

Moreover, there is a transition between these two places; the annotations of the two arcs are [x], where x is a variable of sort data. This variable is defined in the declaration of the implementation; the declaration makes use of the imported sort symbol data. The bracket notation around variable x indicates that the arc expression denotes a multiset with a single element[3] bound to variable x.

Note that both places in the module implementation have *type* data, which exactly corresponds to the type of the import and export places in the interface.

2.2 Module instances

Next, we build a simple system from the module Channel. Figure 3 shows the use of three *instances* of the module Channel. The instances are named ch1, ch2, and

[2] Actually, the keywords import and export, as well as the graphical notation for import and export nodes, are not the point of this paper.

[3] Often, the notation 1`x is also used, e.g., for coloured nets in CPNTools [6].

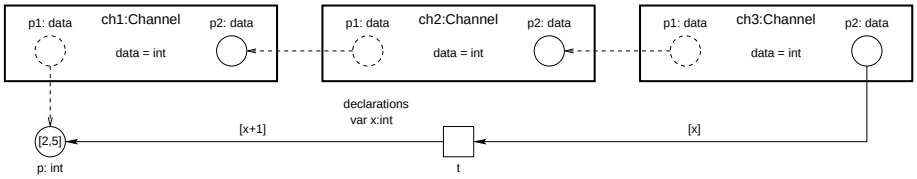


Fig. 3. Using instances of module Channel

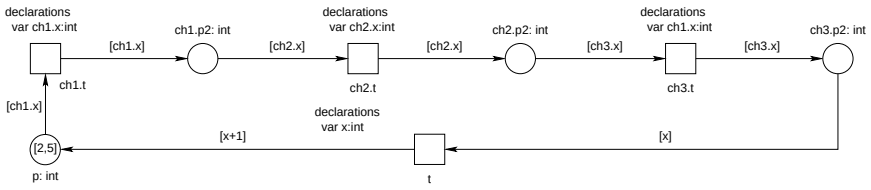


Fig. 4. The resulting model

ch3, respectively. To indicate the instantiation, we use the name of the instance followed by the name of the module definition (inspired by UML).

With the help of this example, we can explain the meaning of import places and import symbols. For each instance of the module, the import place needs to refer to some place outside the module, which will be the one imported for that instance. This is indicated by dashed arrows from the use of the interface to some other parts of the system. Note that export nodes of module instances are seen outside a module. Hence, they can be referred to from import nodes. This way, we get a sequence of three channels. Once the data from the leftmost place is transmitted to the right-most channel, the additional transition increments the token value and sends it back to the start place.

This is where the import symbol data representing a sort comes in again. For each instance of the module Channel, we must provide a sort for the symbol data. In this example, we use the sort int, which is a built-in sort of high-level Petri nets. This way, the chain of channels transmits integer values. However, we could have used any other built-in or user-defined sort for that.

From the model in Fig. 3 and the definition of the module Channel as shown in Fig. 1, the actual Petri net defined is the one shown in Fig. 4. It is obtained by making a copy of the module implementation for each module instance and by merging the nodes identified by the references. Moreover, every occurrence of the sort data in the module implementation is now replaced by the sort it is assigned in this instance of the module: int in our example.

2.3 More advanced concepts

In the rest of this paper, we will formalise these ideas. The formal definitions will however be more general. In our example, we had only import and export

places. In the general definition, there will be also import and export transitions, with basically the same mechanism: fusing the respective transitions.

Moreover, modules can also import operation symbols. Figure 2 shows an example. For some operation $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and some value y of sort B, which is put to the import place, it calculates a pair (x, y) such that $\mathsf{f}(x) = y$ and puts this pair to the output place—if such a pair exists: it magically computes the inverse of f. Note that the module is independent of a particular operator f; it works for any operator, which will be provided when the module is instantiated.

In addition to the import of an operation symbol, this example shows another important feature. In the implementation of the module, we make use of the imported sorts A and B and build a new one, the product sort. This is, actually, one of the technically tricky issues of the formalisation.

Finally, our formalisation allows for exporting sort and operation symbols.

3 Basic Definitions

In this section, we formalise algebraic Petri nets and all the pre-requisites. We introduce the standard concepts of algebraic specifications [7] and of algebraic Petri nets [8–11, 1], but in a notation easing the definition of modules.

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 $MS(A)$.

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 their underlying 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*. 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}*.*

For some set of symbols A and some signature $SIG = (S, O)$ the *restriction* of SIG to symbols in A is $SIG|_A = (S \cap A, O \cap A)$. Note that $SIG|_A$ is not always a signature (e. g. if A contains an operation of O but not the operation sorts).

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 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, $\mathcal{A}$ 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* $\mathcal{A}'|_{S^{SIG} \cup O^{SIG}} = \mathcal{A}$*, written* $\mathcal{A} \subseteq \mathcal{A}'$*.*

3.3 Variables and terms

Let $SIG = (S, O)$ be a signature. An S-indexed set X is a set of *SIG-variables*, if X is disjoint from O. From the set of operations O of the signature and of variables X, *terms* of some sort s can be constructed inductively:

Definition 6 (Terms). *The sets 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 $\mathbb{T}_s^{SIG} = \mathbb{T}_s^{SIG}(\emptyset)$.

Note that, in practice, terms are 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 formal definitions.

Definition 7 (Compatible mapping). *Let SIG and SIG' be two signatures and σ a signature homomorphism from SIG to SIG'. Let X be a set of SIG-variables and X' be a set of SIG'-variables. A mapping $\xi : X \to X'$ is said to be* compatible *with σ if, for every variable $x \in X_s$, we have $\xi(x) \in X'_{\sigma(s)}$. The mappings σ and ξ can be canonically extended to a mapping $\overline{\sigma \cup \xi} : \mathbb{T}^{SIG}(X) \to \mathbb{T}^{SIG'}(X')$ by*

- $\overline{\sigma \cup \xi}(x) = \xi(x)$ *for every* $x \in X$, *and*
- $\overline{\sigma \cup \xi}((o, t_1, \ldots, t_n)) = (\sigma(o), \overline{\sigma \cup \xi}(t_1), \ldots, \overline{\sigma \cup \xi}(t_n))$, *for every operation symbol* $o \in O_{s_1 \ldots s_n s_{n+1}}$, *and, for all terms* $t_k \in \mathbb{T}^{SIG}_{s_i}(X)$.

3.4 Generators

In high-level nets and high-level net modules in particular, we often have some sorts provided, and we need to construct other sorts from them in a standard way. For example, for a given sort s, we need a sort that represents the multiset sorts over that sort, $ms(s)$. We may also want to build the product sort over some sorts (see Fig. 2 for an example). 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 $\mathcal{A}(s)$.

In module definitions, we also want to import sorts to be used in the module implementation without yet knowing which concrete set will be associated with it, since this will only be known when the module is instantiated. Still, we would like to use these sorts and sorts built from them in the module definition. 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 also use them for defining the standard sorts, such as *bool*, along with their operations.

Definition 8 (Generator). *A* generator $G = (GS, GA)$ *consists of*

- *a* sort generator function *GS that, for any given signature $SIG = (S, O)$, returns a signature $GS(SIG) = (S', O')$ such that $S \subseteq S'$ and $O \subseteq O'$; $GS(SIG)$ 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 $\mathcal{A}$, i.e., $\mathcal{A} \subseteq GA(\mathcal{A})$.*

Throughout this paper, we will use a single generator[4] $G = (GS, GA)$, which will be defined in this section. The basic idea is to include, in addition to the

[4] This is a very minimalistic version; there could be many more built-in sorts, generated sorts, and operations (see ISO/IEC 15909-2 [4]); but this is beyond the scope of this paper; our module concept will work for any generator extending this one.

existing sorts, 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, 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 tupling operation $((), s_1, \ldots, s_n)$, and the operation $((], s), s, \ldots, s)$ which makes a multiset out of a list of elements.

Definition 9 (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. *$(not, (bool)) \in O'_{(bool)(bool)}$,*
  4. *$(and, (bool), (bool)), (or, (bool), (bool)) \in O'_{(bool)(bool)(bool)}$,*
  5. *$((], s), s, \ldots, s) \in O'_{s \ldots s(ms,s)}$ for every sort $s \in S'$, where the number of s is the same in both constructs,*
  6. *$(+, (ms, s), (ms, s)) \in O'_{(ms,s)(ms,s)(ms,s)}$ for every $s \in S'$, and*
  7. *$((), s_1, \ldots, s_n) \in O'_{s_1 \ldots s_n(\times, s_1, \ldots, s_n)}$ for all $s_1, \ldots, s_n \in S'$.*

Definition 10 (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. *$GA(\mathcal{A})((not, bool)) = \neg$, where $\neg$ is the boolean negation function,*
  4. *$GA(\mathcal{A})((and, bool, bool)) = \wedge$ and $GA(\mathcal{A})((or, bool, bool)) = \vee$, where $\wedge$ and $\vee$ are the boolean conjunction and disjunction functions,*
  5. *$GA(\mathcal{A})(((], s), s, \ldots, s))(a_1, \ldots, a_n) = [a_1, \ldots, a_n]$, for 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. *$GA(\mathcal{A})(+, (ms, s), (ms, s))) = +$ for every sort $s \in S'$, where $+$ denotes the addition of two multisets over $GA(\mathcal{A})(s)$, and*

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.*

Note that, to avoid overly complex mathematics, we assume that all the symbols used in a basic signature SIG are disjoint from symbols introduced by the generators $GS(SIG)$. We assume that the symbols 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 just needed 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 11 (Sort 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:*

- 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. $\sigma^G((not, (bool))) = (not, (bool))$,
  4. $\sigma^G((and, (bool), (bool))) = (and, (bool), (bool))$, *and* $\sigma^G((or, (bool), (bool))) = (or, (bool), (bool))$,
  5. $\sigma^G(([], s, \ldots, s)) = (([], \sigma^G(s)), \sigma^G(s), \ldots, \sigma^G(s))$ *for every sort* $s \in S$,
  6. $\sigma^G((+, (ms, s), (ms, s))) = (+, (ms, \sigma^G(s)), (ms, \sigma^G(s)))$ *for every sort* $s \in S$, *and*
  7. $\sigma^G(((), s_1, \ldots, s_n)) = ((), \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for all* $s_1, \ldots, s_n \in S$.

In the following, we even use the symbol σ instead of σ^G.

3.5 Nets, algebraic net schemes, and algebraic nets

Now we are prepared to define the basic concepts of this paper.

Definition 12 (Net). *A* net $N = (P, T, F)$ *consists of two disjoint sets* P *and* T *and a set of arcs* $F \subseteq (P \times T) \cup (T \times P)$.

For a clear separation between syntax and semantics, we distinguish between *algebraic net schemes* and *algebraic nets*.

Definition 13 (Algebraic net scheme). *An* algebraic net scheme *is a tuple* $\Sigma = (N, SIG, X, sort, l, c, m)$ *consisting of:*

1. *a* net $N = (P, T, F)$,
2. *a* signature SIG,
3. *a set of* $GS(SIG)$-variables X,
4. *a* place sort *mapping* $sort : P \to S^{GS(SIG)}$,

5. *an* arc label *mapping* $l : F \to \mathbb{T}^{GS(SIG)}(X)$ *such that:*
   - *for all* $(p,t) \in F \cap (P \times T) : l((p,t)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(X)$
   - *for all* $(t,p) \in F \cap (T \times P) : l((t,p)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(X)$,
6. *a* transition condition *mapping* $c : T \to \mathbb{T}^{GS(SIG)}_{(bool)}(X)$,
7. *an* initial marking $m : P \to \mathbb{T}^{GS(SIG)}$ *such that,* $m(p) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}$ *for every place* $p \in P$.

Definition 14 (Algebraic net). *An* algebraic net $(\Sigma, \mathcal{A})$ *is an algebraic net scheme* Σ *equipped with a SIG-algebra* $\mathcal{A}$.

In this paper, we focus on the definition of modules and how they can be used to define other modules. We are not so much interested in their actual behaviour. Therefore, we do not define the firing rule for algebraic nets here. A formalisation of the abstraction in terms of the behaviour of a module is an interesting endeavour—but much beyond the scope of this paper.

4 Modules interfaces and implementation

In this section, we formalise the notion of module interfaces and their implementation, informally introduced in Sect. 2. The module interface describes which places and transitions are imported or exported, and which sort and operation symbols are imported or exported on a purely syntactical level. Moreover, the interface defines the sort of each of the import and export places.

Definition 15 (Module interface).
A module interface $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO})$ *consists of two signatures* SIG_I *and* SIG_O *with disjoint sets of symbols, four pairwise disjoint sets* P_I, T_I, P_O, T_O *and a mapping* $sort_{IO} : P_I \cup P_O \to S^{GS(SIG_I \cup SIG_O)}$.

We call (SIG_I, P_I, T_I) the *import interface*, SIG_I the *imported signature*, P_I the *imported places*, and T_I the *imported transitions*. We call (SIG_O, P_O, T_O) the *export interface*, SIG_O the *exported signature*, P_O the *exported places*, and T_O the *exported transitions*. The mapping $sort_{IO}$ assigns a sort to every place of the interface. Note that this can be any sort that can be generated from the sorts of the import and export signatures.

Definition 16 (Module implementation).
Let $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO})$ *be a module interface. Then, a* module implementation $\mathcal{M} = (\mathcal{I}, \Sigma, \mathcal{A})$ *of interface* $\mathcal{I}$*, consists of:*

1. *the* interface $\mathcal{I}$ *itself,*
2. *an* algebraic net scheme $\Sigma = (N, SIG, X, sort, l, c, m)$ *with* $N = (P, T, F)$ *where* $SIG = (S, O)$ *extends* SIG_I *and* SIG_O *such that* SIG *restricted to the non-imported part,* $SIG \setminus SIG_I = SIG|_{S \setminus S^{SIG_I} \cup O \setminus O^{SIG_I}}$*, is a signature,* $P \supseteq P_I \cup P_O$, $T \supseteq T_I \cup T_O$, *and* $sort \supseteq sort_{IO}$,
3. *a* $SIG \setminus SIG_I$-algebra $\mathcal{A}$.

Note that this definition does not require that $\mathcal{A}$ is a SIG-algebra since some symbols from SIG are imported from SIG_I. The interpretation of the symbols from SIG_I will come from the imported symbols when the module is instantiated. In order to assign the meaning to the remaining symbols, we require $SIG \setminus SIG_I$ to be a signature, and $\mathcal{A}$ to be a $SIG \setminus SIG_I$-algebra. In general, $(\Sigma, \mathcal{A})$ is not an algebraic net; however, it is an algebraic net if SIG_I is empty.

5 Modules definitions and implementations

Up to now, we have defined module interfaces and their implementation. The implementation was given by a monolithic algebraic net. The purpose of modules, however, is to use instances of some modules for defining a system or other modules, which we call *module definitions*. In this section, the notion of module definition as well as its meaning is formalised.

5.1 Module definition

Let $\mathcal{J}$ be a set of module interfaces, which can then be used for defining another module. For each $\mathcal{I} \in \mathcal{J}$, we denote: $\mathcal{I} = ((SIG_I^{\mathcal{I}}, P_I^{\mathcal{I}}, T_I^{\mathcal{I}}), (SIG_O^{\mathcal{I}}, P_O^{\mathcal{I}}, T_O^{\mathcal{I}}), sort_{IO}^{\mathcal{I}})$, $SIG_I^{\mathcal{I}} = (S_I^{\mathcal{I}}, O_I^{\mathcal{I}})$ and $SIG_O^{\mathcal{I}} = (S_O^{\mathcal{I}}, O_O^{\mathcal{I}})$.

First, we introduce a notation for *module instances* resp. the *use* of modules.

Definition 17 (Module instances and uses). *For some $n \in \mathbb{N}$ and for every $k \in \{1, \ldots, n\}$, let $\mathcal{I}_k \in \mathcal{J}$. Then, $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$ is a set of n* module instances *of $\mathcal{J}$. The set $\mathcal{U}$ is called the* module uses.

Note that the interfaces $\mathcal{I}_k$ are not required to be different since the same module may be used multiple times in another module definition. Therefore, in order to be able to distinguish different instances of the same module, a different number is associated with each of them. This is the case in the introductory example of Sect. 2, where three copies of the Channel module are used.

Definition 18 (Module definition). *A* module definition $\mathcal{D} = (\mathcal{I}, \Sigma, \mathcal{U}, (si_k)_{k=1}^n, (pi_k)_{k=1}^n, (ti_k)_{k=1}^n, (so_k)_{k=1}^n, (po_k)_{k=1}^n, (to_k)_{k=1}^n, \mathcal{A})$ *for interface $\mathcal{I}$ over some module interfaces $\mathcal{J}$, consists of the following:*

1. *its own* interface $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO})$,
2. *an* algebraic net scheme $\Sigma = (N, SIG, X, sort, l, c, m)$ *with signature* $SIG = (S, O)$ *and net* $N = (P, T, F)$,
3. *a set of* module instances $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$ *of* $\mathcal{J}$,
4. *for each* $k \in \{1, \ldots, n\}$,
   (a) *a signature homomorphism* $si_k : SIG_I^{\mathcal{I}_k} \to SIG$,
   (b) *an injective signature homomorphism* $so_k : SIG_O^{\mathcal{I}_k} \to SIG \setminus SIG_I$,
   (c) *a mapping* $pi_k : P_I^{\mathcal{I}_k} \to P$,
   (d) *an injective mapping* $po_k : P_O^{\mathcal{I}_k} \to P \setminus P_I$,

(e) a mapping $ti_k : T_I^{\mathcal{I}_k} \to T$,
(f) an injective mapping $to_k : T_O^{\mathcal{I}_k} \to T \setminus T_I$,
such that the co-domains of all homomorphisms so_k *are pairwise disjoint, the co-domains of all mappings* po_k *are pairwise disjoint, and the co-domains of all mappings* to_k *are pairwise disjoint. Moreover, for every* k *and for every* $p \in P_I^{\mathcal{I}_k}$, *we have* $si_k(sort_{IO}^{\mathcal{I}_k}(p)) = sort(pi_k(p))$, *and for every* $p \in P_O^{\mathcal{I}_k}$, *we have* $so_k(sort_{IO}^{\mathcal{I}_k}(p)) = sort(po_k(p))$, *such that* $SIG_D = (S', O')$ *with* $S' = S \setminus (S^{SIG_I} \cup \bigcup_{k=1}^{n} so_k(S^{SIG_O^{\mathcal{I}_k}}))$ *and* $O' = O \setminus (O^{SIG_I} \cup \bigcup_{k=1}^{n} so_k(O^{SIG_O^{\mathcal{I}_k}}))$ *is a signature, and*

5. $\mathcal{A}$ *is a* SIG_D*-algebra.*

Basically, the module definition consists of an algebraic net scheme Σ, where the homomorphisms and mappings (see condition 4) from the interfaces of the used module instances $\mathcal{U}$ to Σ indicate how the instances of the modules are embedded into Σ. This concerns the embedding of places and transitions as well as the use of the different symbols of the signatures. Note that if an export symbol of a module instance is mapped to a symbol of Σ, this symbol will get its meaning from this module instance. Therefore, condition 4 requires that these mappings do not overlap. The meaning of the symbols of the import signature will be defined when the module is used; therefore, the module definition itself does not need to give a definition to these symbols. Therefore, the algebra $\mathcal{A}$ does not need to assign a meaning for the symbols coming from the import signature of the defined module or from the export symbols of the used modules. The remaining part of the signature, denoted by SIG_D in the above definition, must be a signature and $\mathcal{A}$ must be a SIG_D-algebra (condition 5).

5.2 Denoted implementation

In this section, we will define the module implementation inferred from a module definition, i.e., based on other modules. Let us consider a module definition

$$\mathcal{D} = (\mathcal{I}, \Sigma, \mathcal{U}, (si_k)_{k=1}^n, (pi_k)_{k=1}^n, (ti_k)_{k=1}^n, (so_k)_{k=1}^n, (po_k)_{k=1}^n, (to_k)_{k=1}^n, \mathcal{A})$$

as defined in Def. 18 using module instances $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$. In order to define the module implementation, the implementations of the used modules must be known. Let us assume that for each $k \in \{1, \ldots, n\}$, $\mathcal{M}_k = (\mathcal{I}_k, \Sigma_k, \mathcal{A}_k)$ is an implementation for interface $\mathcal{I}_k$. The basic idea of the module defined by $\mathcal{D}$ is to make a disjoint union of all signatures and nets of the implementations and the module definition itself, and to transform the arc, place, and transition labels accordingly. However, some parts need to be identified, as defined by the homomorphism between the signatures and the mappings from the interface places and transitions to the places and transitions of the module definition.

We start with defining the signature of the module implementation, which will be denoted with $\widehat{SIG}$. First, we summarise the available signatures:

1. The signature $SIG = (S, O)$ from the module definition.

2. For every use of a module $(k, \mathcal{I}_k)$, there are two disjoint signatures $SIG_I^{\mathcal{I}_k}$ and $SIG_O^{\mathcal{I}_k}$. We define $SIG^{\mathcal{I}_k} = SIG_I^{\mathcal{I}_k} \cup SIG_O^{\mathcal{I}_k}$. Moreover, there is a signature homomorphism $f_k : SIG^{\mathcal{I}_k} \to SIG$ defined by $f_k = so_k \cup si_k$.
3. For every use of a module $(k, \mathcal{I}_k)$, the implementation $\mathcal{M}_k = (\mathcal{I}_k, \Sigma_k, \mathcal{A}_k)$ has a signature SIG_k and variables X_k. By definition $SIG^{\mathcal{I}_k} \subseteq SIG_k$ holds.

Definition 19 (Signature $\widehat{SIG}$ and homomorphisms σ_k). *For each $k \in \{1, \ldots, n\}$, the mapping σ_k is a signature homomorphism from SIG_k to $\widehat{SIG}$, defined as follows: for every $x \in SIG_k$ for which $f_k(x)$ is defined: $\sigma_k(x) = f_k(x)$; for every other symbol $x \in SIG_k$: $\sigma_k(x) = (k, x)$. We define $\widehat{SIG} = (\widehat{S}, \widehat{O})$ by $\widehat{S} = S \cup \bigcup_{k=1}^{n} \sigma_k(S^{SIG_k})$ and $\widehat{O} = O \cup \bigcup_{k=1}^{n} \sigma_k(O^{SIG_k})$, where the arities carry over by interpreting σ_k as a signature homomorphism.*

Note that, in the definition of $\sigma_k(x)$, the pair (k, x) is used to make this symbol of SIG_k different from all the other symbols. The signature $\widehat{SIG}$ will be the signature of the defined module implementation. The signature homomorphisms σ_k relate the signatures of the implementations to $\widehat{SIG}$; they will be used to transfer the labels from the different module implementations to the defined module implementation.

The variables from the different modules are made disjoint in the same way.

Definition 20 (Variables $\widehat{X}$). *For every $k \in \{1, \ldots, n\}$, the mapping ξ_k is defined by: $\xi_k(x) = (k, x)$ for every variable $x \in X_k$. The set of all variables is defined by $\widehat{X} = X \cup \bigcup_{k=1}^{n} \xi_k(X_k)$.*

The meaning of the non-imported symbols of $\widehat{SIG}$ is defined by an $\widehat{SIG} \setminus SIG_I$-algebra $\widehat{\mathcal{A}}$. Basically, this meaning carries over from the other algebras via the respective homomorphisms.

Definition 21 (Algebra $\widehat{\mathcal{A}}$). *The $\widehat{SIG} \setminus SIG_I$-algebra $\widehat{\mathcal{A}}$ associated with $\mathcal{A}$ is defined as follows: If $\mathcal{A}(x)$ is defined, then $\widehat{\mathcal{A}}(x) = \mathcal{A}(x)$; if $\mathcal{A}_k(x)$ is defined, then $\widehat{\mathcal{A}}(\sigma_k(x)) = \mathcal{A}_k(x)$.*

By the conditions imposed on the algebras and the signature homomorphisms, this definition of $\widehat{\mathcal{A}}$ is unique and it is a $\widehat{SIG} \setminus SIG_I$-algebra.

For experts, $\widehat{SIG}$ and $\widehat{\mathcal{A}}$ are pushout constructions in an appropriate category; but the categorical constructions are beyond the scope of this paper.

Next, we define the places and transitions of the module implementation, which are basically a disjoint union of all the places and transitions of the used module implementations and the places and transitions of the module definition itself. The places and transitions identified by the mappings from the import and export interfaces will be merged. First, we summarise what we already know:

1. The net $N = (P, T, F)$ of the module definition. The set of all nodes of that net is $Z = P \cup T$.
2. For every use of a module $(k, \mathcal{I}_k)$, let $Z^{\mathcal{I}_k}$ be the set of nodes of the interface and let Z_k be the set of nodes of the implementation, $Z^{\mathcal{I}_k} \subseteq Z_k$. There is a mapping $g_k : Z^{\mathcal{I}_k} \to Z$, which is defined by $g_k = pi_k \cup po_k \cup ti_k \cup to_k$.

Definition 22 (Places $\widehat{P}$ and transitions $\widehat{T}$). *For every $k \in \{1, \dots n\}$, a mapping e_k is defined as follows: $e_k(x) = g_k(x)$ for every $x \in Z^{\mathcal{I}_k}$, and $e_k(x) = (k, x)$ for every $x \in Z_k \setminus Z^{\mathcal{I}_k}$. The set of places of the module implementation defined by the module definition is defined by $\widehat{P} = P \cup \bigcup_{k=1}^{n} e_k(P_k)$ and the set of transitions is defined by $\widehat{T} = T \cup \bigcup_{k=1}^{n} e_k(T_k)$.*

Now we have all the ingredients for defining the module implementation. Basically, the mappings of the module instances carry over from the module implementations via the homomorphism:

Definition 23 (Defined module implementation).
Let $\mathcal{D} = (\mathcal{I}, \Sigma, \mathcal{U}, (si_k)_{k=1}^n, (pi_k)_{k=1}^n, (ti_k)_{k=1}^n, (so_k)_{k=1}^n, (po_k)_{k=1}^n, (to_k)_{k=1}^n, \mathcal{A})$ be a module definition with module uses $\mathcal{U} = \{(1, \mathcal{I}_1), \dots, (n, \mathcal{I}_n)\}$ and module implementations $\mathcal{M}_k = (\mathcal{I}_k, \Sigma_k, \mathcal{A}_k)$ for each k.

The module implementation defined by $\mathcal{D}$ is $\widehat{\mathcal{M}} = (\mathcal{I}, \widehat{\Sigma}, \widehat{\mathcal{A}})$ where $\widehat{\Sigma} = (\widehat{N}, \widehat{SIG}, \widehat{X}, \widehat{sort}, \widehat{l}, \widehat{c}, \widehat{m})$ with $\widehat{N} = (\widehat{P}, \widehat{T}, \widehat{F})$ such that
$\widehat{F} = F \cup \bigcup_{k=1}^{n} \{(e_k(x), e_k(y)) \mid (x, y) \in A_k\}$.

The mappings $\widehat{l}$, $\widehat{sort}$, $\widehat{c}$, and $\widehat{m}$ are defined as follows:

- *$\widehat{l}(f) = l(f)$ for every arc $f \in F$ and $\widehat{l}(f) = \overline{\sigma_k \cup \xi_k}(l_k(f))$ for every arc $f \in F_k$.*
- *$\widehat{sort}(p) = sort(p)$ for every place $p \in P$ and $\widehat{sort}(e_k(p)) = \sigma_k(sort_k(p))$ for every place $p \in P_k$.*
- *for every transition $t \in T$, for which there exists no k with $t \in e_k(T_O^k)$, we define $\widehat{c}(t) = c(t)$; for every transition $t \in T^k \setminus T_I^k$ we define $\widehat{c}(e_k(t)) = \overline{\sigma_k \cup \xi_k}(c_k(t))$.*
- *for every place $p \in P$, for which there exists no k with $p \in e_k(P_O^k)$, we define $\widehat{m}(p) = m(p)$; for every place $p \in P^k \setminus P_I^k$ we define $\widehat{m}(e_k(p)) = \overline{\sigma_k \cup \xi_k}(m_k(p))$.*

As mentioned earlier, $\widehat{\mathcal{A}}$ is a $\widehat{SIG} \setminus SIG_I$-algebra. By the conditions imposed on the module definitions, $\widehat{l}$, $\widehat{sort}$, $\widehat{c}$, and $\widehat{m}$ are properly defined. Altogether, the defined module implementation is uniquely defined:

Theorem 1. *For an interface $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO}^{\mathcal{I}})$ and a module definition $\mathcal{D}$ for $\mathcal{I}$ with module uses $\mathcal{U} = \{(1, \mathcal{I}_1), \dots, (n, \mathcal{I}_n)\}$ and module implementations $\mathcal{M}_k$, $\widehat{\mathcal{M}} = (\mathcal{I}, \widehat{\Sigma}, \widehat{\mathcal{A}})$ is a uniquely defined module implementation. If SIG_I is empty, then $(\widehat{\Sigma}, \widehat{\mathcal{A}})$ is an algebraic net.*

6 Example

Here, we present an example of a railway case study described in a modular way in [12]. It is now slightly changed so as to be consistent with our notations.

The example models a toy railway composed of several track sections, as shown in Fig. 5, either connected directly or via a switch. Several trains can

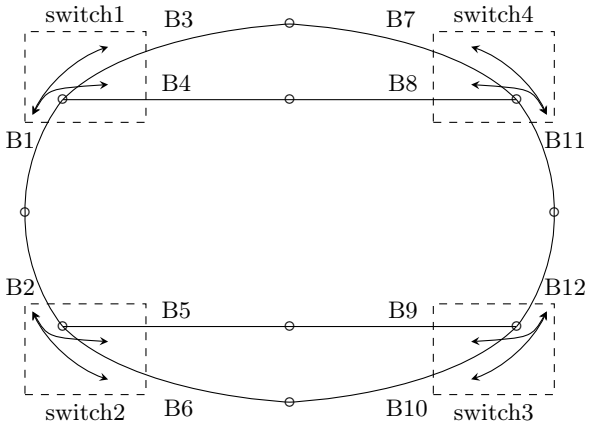


Fig. 5. The tracks of the model railway.

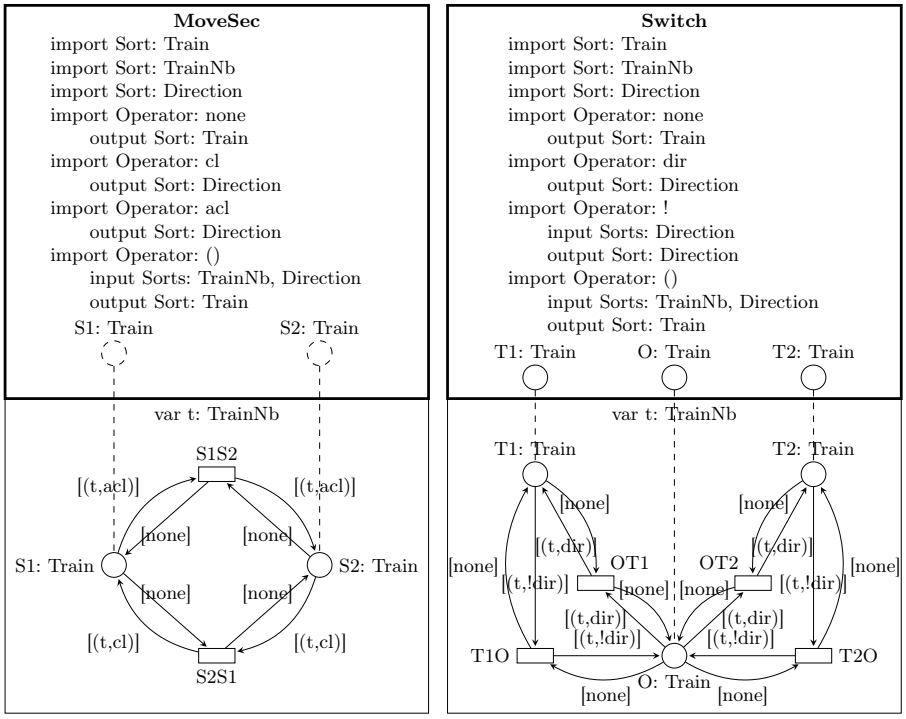
circulate at the same time, and the routing policy of trains should ensure that there is no collision and the system is always running. The modular design of such a system was the scope of [12] and lead to identifying 2 modules: **MoveSec** models the moves between two directly connected tracks while **Switch** is a switch connecting three track sections. In both modules, each place corresponds to a track section, which may or may not be occupied by a train. The transitions reflect the possible moves. These two modules are depicted in Fig. 6 and are used by a top-level module, which captures the whole system, in Fig. 7.

We have chosen to define the track sections within the **Switch** modules since the switch is the most elaborate part of the system. Therefore the places in module **Switch** exports its places. Conversely, module **MoveSec** imports its places, as they are defined elsewhere.

Other choices in this particular example could have been made for import and export places. For instance, the tracks of a toy railway are asymmetric since for connecting pieces, one side of a track gets inside (the other side of) another track. This easily fits an imported place and an exported place scheme for the **MoveSec** module. But this also leads to two types of switch modules: one exporting places `T1` and `T2` and importing `0`, and the other doing the converse.

The choice we made illustrates parameterisation of modules. Note that module **Switch** imports a direction operator **dir**, which allows for using the same module to represent all switches, even though they operate in symmetrical ways. The operator is instantiated when connecting the **Switch** module, as shown in Fig. 7. It then takes value `cl` for switches `switch1` and `switch3`, and value `acl` for switches `switch2` and `switch4`.

Finally, the sorts and operators are defined in the top-level module and can be used consistently by all modules. This can be considered as a *global* definition. A `Train` on a track section is identified by a `TrainNb`, and a can move in a given



(a) The `MoveSec` module (b) The `Switch` module

Fig. 6. The railway example modules.

`Direction`. The operators consist of 3 constants: `none` indicating that no train is on a track section, `cl` and `acl` giving the possible directions of trains (clockwise and anticlockwise). The unary operator `!` is intended to change a direction into the opposite one. Finally, `()` forms a pair with a train identity and a direction in which the train moves, detailing a train present on a track section.

7 Discussion and extensions

In this section, we briefly discuss our module concept and some issues that should be considered for the work on Part 3 of the ISO/IEC-15909 standard.

7.1 Abstraction and refinement

One of the main objectives of using modular design is to handle abstraction and refinement features. Our proposal fits with such a scheme by separating the module definition which lies at an abstract level and the module implementation. Moreover, modules can import constructs from others and provide constructs to be used by others.

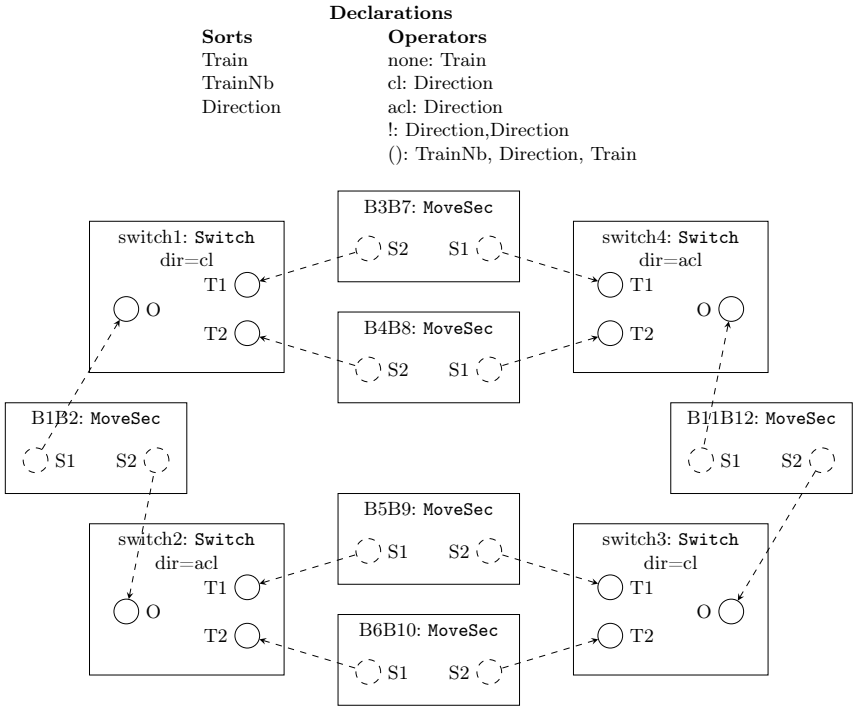


Fig. 7. The top-level net model of the model railway.

Refinement can be pursued further, by detailing the functioning of a module through other new modules. To cope with such a process, it will be most helpful to provide a *hierarchy of modules*, showing how they are embedded in one another. The current module definitions allow us to build modules in a hierarchical way. However, for practical use, a designer should be provided with a view of the hierarchy (as in e.g., Hierarchical CPNs).

The example of Sect. 6 also shows that *parameterisation* of modules is possible. This is a key feature for reuse of modules in different contexts.

7.2 Aggregation of label information

In our formalisation, some annotations of places and transitions are ignored. For example, the initial marking of a place is always taken from the module where the place is actually defined. If a module imports a place, the module can define an initial marking. But this marking is irrelevant since it will come from wherever the imported place is defined. The same holds for the transition condition.

For the transition condition, it might make sense to use a conjunction of all transition conditions attached to the transition. As concerns the initial marking, it might make sense to use the sum of all initial markings. Since we started from the modular PNML semantics, we did not include that here.

Since such an *aggregation mechanism* seems to be reasonable in at least some cases, aggregation should be considered for the upcoming standard. However, this

will introduce some technical difficulties. Not all annotations can be aggregated in a reasonable way: clearly the aggregation function would require an associative and commutative operation with a neutral element for making the aggregation independent from a specific order. Even if the operation is associative and commutative, its syntactical representation is not. Therefore, there would not be a canonical syntactical representation for the defined module implementation.

The aggregation mechanism could be even more advanced. For example, the defining module could provide the operation that is used for the aggregation. This way, it would be up to the defining module to decide whether and how particular labels of modules using it should be aggregated. What is reasonable, necessary, easily usable, and semantically sound is subject to future research.

7.3 Export of variables

In our formalisation, modules can export and import only sort and operation symbols. It does not allow for exporting variables. In the case of synchronous communication via merging of transitions, it might, however, make sense to use a common variable for such transitions to exchange values between different partners during a synchronisation. Therefore, it might be worthwhile to also export and import some of the variables along with a transition.

A formalisation, however, requires that variables are defined locally to a transition as for example proposed by Schmidt [13]. The formalisation is a bit more technical, but we believe that this concept should be included in the standard.

7.4 Node connection policies

In our definition of export and import nodes, the other modules could connect to that node as to any other node. In some cases, some uses might not be intended at all. In our introductory example from Fig. 1, it does not make much sense for a module using the Channel module to add a token to export place p2. Though adding a token does not do much harm here, the module might want to restrict the use of this place so that tokens can only be removed from that place. Right now such a restriction cannot be enforced and would just be a textual recommendation of the use of a node.

It would be nice if a module could provide some composition policies that state in which way a node may be used, in order to define and to enforce communication paradigms. What exactly should be expressible by such policies and how a language for expressing such policies should look like, requires further investigation.

7.5 Generators

The key mechanism for having the module concept work is the generator. This way, it is possible to construct standard sorts out of existing sorts without even knowing the underlying algebra yet.

Up to now, there is only one fixed generator, which supports the standard generic constructs on sorts like multisets or products over sorts. It is not yet possible to define user-defined generic constructs. Of course, it would be useful to allow the extension of this generator within a module definition, so that a module could define new generic constructs. To this end, we could use existing theory from algebraic specifications. The question, however, is how much expressivity is needed and worth the effort to be included in the standard.

The idea of generators could also serve a different purpose: As we have seen, we used the generator for defining the built-in sorts and the standard constructs. Actually, many variants of high-level Petri nets differ only in these standard sorts and constructs. One example are well-formed nets [14], which are currently included as a special version in Part 1 of ISO/IEC 15909 (renamed symmetric nets). Generators could ease the definition of sub-classes of high-level Petri nets.

8 Related work

Many modular constructs have been proposed in the literature. Our aim is to propose a framework capturing most of these mechanisms. In this section, we show how such mechanisms are dealt with.

Our approach extends the work in [5] by providing a formal and flexible definition. The communication mechanisms proposed in [15] are place fusion and transition fusion. They are easily handled by place and transition import/export features. The main difference with our proposal is the asymmetry between importing and exporting, whereas plain fusion is symmetric. But this is no restriction.

One of the earliest and most widespread modular approach is Hierarchical Coloured Petri Nets [16] and their implementation within CPNTools [6]. They also use the concept of port places, which can be defined as input, output or both. The structuring of nets is presented via a hierarchy of modules. Such nets also use the place fusion concept, which is captured by our proposal.

9 Conclusion

In this paper, we have shown that there is a formal foundation for the concepts of modular PNML. We also identified some issues that should be considered and resolved in the standardisation of the module concept in Part 3 of ISO/IEC 15909. All kinds of proposals, suggestions, and concerns are most welcome—as is any active participation in the standardisation process.

Acknowledgements We would like to thank Anne Haxthausen and Hubert Baumeister for some helpful discussions on the category theory constructions behind the concepts of this paper. We hope that we, eventually, will write a joint paper from that perspective.

References

1. ISO/IEC: Software and Systems Engineering – High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909 (2004)
2. Kindler, E., Weber, M.: A universal module concept for Petri nets – an implementation-oriented approach. Informatik-Bericht 150, Humboldt-Universität zu Berlin, Institut für Informatik (2001)
3. Weber, M., Kindler, E.: The Petri Net Markup Language. In Ehrig, H., Reisig, W., Rozenberg, G., Weber, H., eds.: Petri Net Technologies for Modeling Communication Based Systems. Volume 2472 of LNCS. Springer (2003) 124–144
4. ISO/JTC1/SC7/WG19: Software and Systems Engineering – High-level Petri Nets, Part 2: Transfer Format. FDIS 15909-2 (under ballot), v. 1.3.6, ISO/IEC (2008)
5. Kindler, E.: Modular PNML revisited: Some ideas for strict typing. In: Proc. AWPN 2007, Koblenz, Germany. (2007)
6. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Journal of Software Tools for Technology Transfer **9**(3-4) (2007) 213–254
7. 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)
8. 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)
9. 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
10. 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
11. Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science **80** (1991) 1–34
12. Choppy, C., Petrucci, L., Reggio, G.: A modelling approach with coloured Petri nets. In: Proc. 13th Int. Conf. on Reliable Software Technologies—Ada-Europe, Venice, Italy. Volume 5026 of LNCS., Springer-Verlag (2008) 73–86
13. 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
14. 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
15. Christensen, S., Petrucci, L.: Modular analysis of Petri nets. The Computer Journal **43**(3) (2000) 224–242
16. Jensen, K.: Coloured Petri Nets: Basic concepts, analysis methods and practical use. Volume 1: basic concepts. Monographs in Theoretical Computer Science. Springer-Verlag (1992)