



Layered Data: A Modular Formal Definition without Formalisms

Alban Linard, Benoît Barbot, Didier Buchs, Maximilien Colange, Clément Démoulins, Lom Messan Hillah, Alexis Martin

► To cite this version:

Alban Linard, Benoît Barbot, Didier Buchs, Maximilien Colange, Clément Démoulins, et al.. Layered Data: A Modular Formal Definition without Formalisms. Petri Nets and Software Engineering (PNSE 2016), Jun 2016, Toruń, Poland. pp.287-306. hal-01353944

HAL Id: hal-01353944

<https://hal.sorbonne-universite.fr/hal-01353944>

Submitted on 16 Aug 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Layered Data: a Modular Formal Definition without Formalisms

Alban Linard¹², Benoît Barbot³, Didier Buchs¹, Maximilien Colange¹²,
Clément Desmoulins⁵, Lom Messan Hillah⁴, and Alexis Martin⁴

¹ Centre Universitaire d’informatique, Computer Science Department,
University of Geneva

² Inria, LSV, ENS Cachan, Université Paris-Saclay, France

³ LACL, Université Paris Est Créteil, France

⁴ Univ. Paris Ouest, and Sorbonne Universités, UPMC Univ Paris 06,
CNRS, LIP6 UMR 7606, 4 place Jussieu F-75005 Paris

⁵ EPITA Research and Development Laboratory (LRDE)

Abstract. Defining formalisms and models in a modular way is a painful task. Metamodeling tools and languages have usually not been created with this goal in mind. This article proposes a data structure, called *layered data*, that allows defining easily modular abstract syntax for formalisms and models. It also shows its use through an exhaustive example. As a side effect, this article discusses the notion of formalism, and asserts that they do not exist as standalone objects, but rather as relations between models.

Keywords: metamodeling, modularity, formalism

1 Introduction

CosyVerif [10] is a software platform dedicated to the modeling and verification of complex systems. It allows its users to create models in various formalisms, such as automata or Petri nets, and run tools on them. It also provides a rarely found feature : the ability to create new formalisms, either derived from the already existing ones, or defined from scratch. Formalisms are used in CosyVerif to generate model parsers and pretty-printers, graphical editors, and to select what tools are available to the modeller.

This nice feature reveals problems in the current use of formalisms, models, and tools in the community of modeling and verification: the ability to define formalisms from “building blocks” (for instance to building the Symmetric Petri nets formalism from expressions and Petri nets formalisms, the latter built from the graph formalism); and the simplicity, maintainability and readability of such definitions. The previous formalism definition language used in CosyVerif [9] still fails to represent complex modular formalisms, even though it has been designed carefully towards this goal.

This article presents a new definition language for formalisms and models to be used within the CosyVerif platform, that seems powerful enough to represent

a wide range of modular formalisms. To do so, we propose another modeling approach called *layered data*. It is a data structure representing trees organized in layers. A flattened view allows walking through the data. We also show how they can be used to represent the abstract syntax for a complex hierarchy of formalisms and models. Starting from no predefined formalisms, and using only a few building mechanisms, we are able to create step-by-step a formalism for Petri net state spaces.

2 Layered data

Layered data are a representation for finite data, structured as directed rooted trees with labels on vertices and edges, and organized in layers. It is suitable, among other things, for configurations (where a user configuration can override parts of a default one), or for the representation of formalisms and models. We present the latter use in this article.

We present the notions of layered data through an informal illustration before the formal definition. The definition itself is given in three steps, each one adding new notions above its predecessor. Definitions should then be interpreted as **redefinitions** of what has been presented before. For the illustration, we choose an expert in trees: Rahan [19].⁶ He is a fictional smart prehistoric man, that travels across the world, learns from people and transmits knowledge to the others.

One day, Rahan comes to a village with a strange tradition. Every villager plants and grows a tree in a clearing (the same for all). During his life, he puts all his knowledge represented as pictures within his own tree, in order to transmit knowledge to other living of future villagers. Trees are organized to help searching: each branch correspond to a domain, and its ramifications to subdomains, and is painted with a descriptive symbol. Pictures of knowledge can be put at any fork or leaf.

In this section, we note:

- \mathbb{B} the set of Boolean values;
- $\mathcal{P}(\mathbb{S})$ the powerset of \mathbb{S} ;
- $\mathcal{L}(\mathbb{S})$ the lists of elements in \mathbb{S} , defined inductively by: the empty list $[] \in \mathcal{L}(\mathbb{S})$, and $\forall s \in \mathbb{S}, \forall l \in \mathcal{L}(\mathbb{S}), s :: l \in \mathcal{L}(\mathbb{S})$; and a convenient notation using square brackets: $[e_1, \dots, e_n] = e_1 :: \dots :: e_n :: []$;
- \oplus a concatenation operator on lists, such that $\forall l \in \mathcal{L}(\mathbb{S}), [] \oplus l = l$, and $\forall s \in \mathbb{S}, \forall l, l' \in \mathcal{L}(\mathbb{S}), (s :: l') \oplus l = s :: (l' \oplus l)$;
- \perp a special value that denotes “no value”; it cannot appear in any list: $\perp :: l = l$, this implies for instance that $[a, \perp, b] = [a, b]$.

The trees can be modeled as:

- \mathbb{L} the domain of labels (drawings) on edges (branches) or vertices (forks or leaves);

⁶ We are very sorry for Tarzan, who did not pass the casting.

- \mathbb{L}^\perp the domain of labels augmented with a special value \perp that means “no value” (no drawing); $\mathbb{L}^\perp = \mathbb{L} \uplus \{\perp\}$;
- \mathbb{V} the set of vertices, defined by: $\langle label, branches \rangle \in \mathbb{V}$ for all $label \in \mathbb{L}^\perp$ and $branches \in \mathbb{L} \rightarrow \mathbb{V}^\perp$; we note $v.label$ (respectively $v.branches$) the value of the *label* (respectively *branches*) part of a vertex $v \in \mathbb{V}$;
- \mathbb{V}^\perp the set of vertices augmented with a special value \perp that means “no vertex”: $\mathbb{V}^\perp = \mathbb{V} \uplus \{\perp\}$.
- the graph formed by vertices in \mathbb{V} and edges given by *branches* must be a forest of trees [11], *i.e.*, a set of disjoint directed rooted trees.

A layered data is a vertex in set \mathbb{V} with two functions for manipulation: *climb* and *get*. Climbing up a knowledge tree is performed by function *climb*. It takes a vertex (or \perp for none) and a label as parameters, and returns the vertex (or \perp for none) reached by following the edge with the corresponding label. Obtaining the value of a vertex (a fork or leaf) is performed by function *get*, that takes a vertex (or \perp for none) and returns its value (or \perp for none).

$$\begin{aligned}
climb &: \mathbb{V}^\perp \times \mathbb{L} \rightarrow \mathbb{V}^\perp \\
climb(\perp, l) &= \perp \\
climb(v, l) &= v.branches(l) \text{ if } v \neq \perp \\
get &: \mathbb{V}^\perp \rightarrow \mathbb{L}^\perp \\
get(\perp) &= \perp \\
get(v) &= v.label \text{ if } v \neq \perp
\end{aligned}$$

A medicine man with two apprentices comes to discuss with Rahan. After his death, both apprentice take his role, and both would like to continue their master’s knowledge tree. Giving the tree to one of them, or to both, is impossible, as it could create conflicts between the two apprentices.







Rahan thinks and proposes a solution. Each apprentice will grow his own tree, but also grow lianas between branches of his trees to branches in their master’s one. Each liana means that the knowledge can be searched within their master’s branch, if it is not found in the apprentice own tree. So, they can only be followed from the apprentices trees to the master’s one. This solution allows each apprentice to complete the knowledge of the medicine man, within their own tree, and even to disagree.

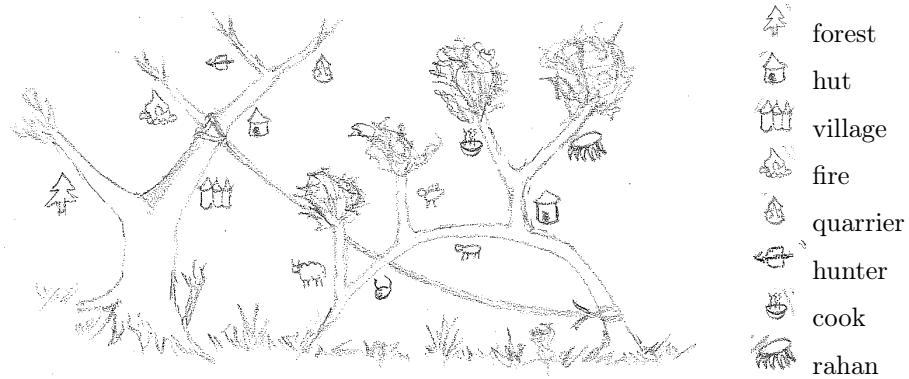
We extend \mathbb{V} the set of vertices to represent the lianas:

- \mathbb{V} the set of vertices, defined by: $\langle label, branches, parents \rangle \in \mathbb{V}$ for all $label \in \mathbb{L}^\perp$ and $branches \in \mathbb{L} \rightarrow \mathbb{V}^\perp$ and $parents \in \mathcal{L}(\mathbb{V})$.

Moreover, as for a single label there may be several paths (one in the current tree, and others by following the liana), we extend the functions *climb* and *get* to work on a list of vertices (sorted by decreasing priority, given by the position in the list) instead of a single one:

$$\begin{aligned}
& climb : \mathcal{L}(\mathbb{V}) \times \mathbb{L} \rightarrow \mathcal{L}(\mathbb{V}) \\
& climb(\perp, l) = \perp \\
& climb(v :: vs, l) = v.branches(l) :: climb(v.parents, l) \oplus climb(vs, l) \\
& get : \mathcal{L}(\mathbb{V}) \rightarrow \mathbb{V}^\perp \\
& get([]) = \perp \\
& get(h :: t) = h.label \text{ if } h.label \neq \perp \\
& get(h :: t) = get(t) \text{ otherwise}
\end{aligned}$$

The picture below illustrates the search within layered data. We are looking for    (Rahan's hut within the village, Rahan begin represented with his famous necklace). Starting from the left tree, a villager takes the branch labeled by , then the branch labeled by , but does not find the symbol for Rahan. The villager looks for a liana, but there is none at this fork. So, he goes backward to the previous fork, and finds a liana. The villager follows it, to the tree on the right. There, he start back climbing with the  symbol, and finds the symbol for Rahan in a branch.



Rahan leaves the village in his quest for the sun's lair, and comes back after some years. The old medicine man has died. His apprentices come to expose a new problem to Rahan. Their old master's tree has two trunks: one for the symptoms and one for the cures. The old man has grown lianas from symptoms to cures in his own tree. The former apprentices have also grown lianas from their trees to their master's branches.

But one of the apprentices has found a better medicine for fractures. Villagers that start climbing his tree usually end within the deceased man's one, where they are then stuck within. They find there only the old cure.

Rahan thinks afresh, and proposes a solution: lianas should not be used within the same tree, but only to refer to another tree. Instead, villagers should

use reference signs, composed of a *notion symbol* telling where to start back climbing, and a sequence of *branch symbols* to follow on branches.

Notion symbols are put at forks or leaves, not on branches. They should give information about the notions represented within that part of the tree, for instance medicine, animals or songs. There can be several notion symbols on one fork (or leaf), for instance some songs are also medicines.

When they encounter a reference sign, villagers must go back to their initial tree, by following lianas in the reverse order. They must then search the fork corresponding to the notion symbol, by going back from the current branch to the trunk. This route ensures that they stay in the most specific use of the notion for the knowledge they search.

When the notion symbol is found, villagers must start again to climb up by following first the sequence of branch symbols given by the sign, and continue by following the symbols they were initially following.

This solution is a bit complex, but after some training, all villagers are able to understand and follow the rules.⁷ One difficulty is that villagers must also follow lianas when looking for notion symbols, as they would do for branch symbols. It creates sometimes lengthy journeys in the trees, that luckily provide fruits to take a break.

The *climb* function now has to remember where it started from, and what labels have been followed. We note:

- \mathbb{R} a set of notion symbols; and \mathbb{R}^\perp the set of notion symbols or \perp for none;
- $\mathbb{V}^r = \mathbb{V}^\perp \uplus \mathbb{R}$ vertices augmented with references;
- $\langle origin, labels \rangle \in \mathbb{P}$ for all $origin \in \mathbb{V}$ and $labels \in \mathcal{L}(\mathbb{L})$, paths starting from a specific vertex and following labels;
- $branches \in \mathbb{L} \rightarrow \mathbb{V}^r$ the *branches* function in vertices is modified to allow returning references.

We extend \mathbb{V} the set of vertices to represent the references:

- \mathbb{V} the set of vertices, defined by: $\langle label, branches, parents, reference \rangle \in \mathbb{V}$ for all $label \in \mathbb{L}^\perp$, $branches \in \mathbb{L} \rightarrow \mathbb{V}^\perp$, $parents \in \mathcal{L}(\mathbb{V})$ and $reference \in \mathbb{R}^\perp$.

The final definition of *climb* is given below. *build* is a utility function that rebuilds recursively all possible vertices from an initial vertex and a list of labels.

$$\begin{aligned}
climb &: \mathbb{P} \times \mathcal{L}(\mathbb{V}) \times \mathbb{L} \rightarrow \mathbb{P} \times \mathcal{L}(\mathbb{V}) \\
climb(p, \perp, l) &= \langle p, \perp \rangle \\
climb(p, v :: vs, l) &= \langle \langle p.origin, l :: p.labels \rangle, dereference(p, v.branches(l)) :: vs_1 \oplus vs_2 \rangle \\
&\quad \text{where } climb(p, v.parents, l) = \langle p_1, vs_1 \rangle \\
&\quad \text{and } climb(p, vs, l) = \langle p_2, vs_2 \rangle
\end{aligned}$$

⁷ In this story, Rahan discovered algorithms, and is thus the first computer scientist!

$$\begin{aligned}
& \text{dereference} : \mathbb{P} \times \mathbb{V}^r \rightarrow \mathcal{L}(\mathbb{V}) \\
& \text{dereference}(p, \perp) = \perp :: [] \\
& \text{dereference}(p, v \in \mathbb{V}) = v :: [] \\
& \text{dereference}(p, r \in \mathbb{R}) = \begin{cases} \langle p, vs_1 \oplus vs_2 \rangle & \text{if } \exists_{v \in vs_1}, v.\text{reference} = r \\ \langle p, vs_2 \rangle & \text{otherwise} \end{cases} \\
& \text{where } p.\text{labels} = l :: ls \\
& \text{and } \text{build}(\langle p.\text{origin}, [] \rangle, [p.\text{origin}], ls) = vs \\
& \text{and } \text{climb}(p, vs, l) = \langle p_1, vs_1 \rangle \\
& \text{and } \text{climb}(\text{dereference}(p, vs), l) = \langle p_2, vs_2 \rangle \\
& \text{build} : \mathbb{P} \times \mathcal{L}(\mathbb{V}) \times \mathcal{L}(\mathbb{L}) \rightarrow \mathbb{P} \times \mathcal{L}(\mathbb{V}) \\
& \text{build}(p, vs, l :: ls) = \text{climb}(\text{build}(p, vs, ls), l) \\
& \text{build}(p, vs, []) = vs
\end{aligned}$$

climb can perform a lot of operations. Its worst case is the search of a non-existing data, as it has to walk through branches and all their parents to detect absence. This article does not provide an analysis of the theoretical complexity. However, implementation is usually quite fast using caching or optimizations such as flattening several layers into one.

A few thousand years later, we are still facing the same problem as Rahan for (meta) modeling systems: knowledge trees have simply been replaced by formalisms or models.

3 What are Formalisms and Models?

Rahan chose simplicity: there were only trees, and then he added the minimal notions required by the needs of the villagers: lianas and reference signs. We can ask ourselves the same question. What are the minimal notions required to represent formalisms and models in a modular way? But also, is it mandatory to have formalisms and models? What is the precise definition of each one?

Papers on formal methods usually make use of the terms “formalism” and “model” without defining them. From their use, and from a small survey around us⁸, we inferred the following definition:

A formalism is a mathematical description for a class of models, like a type or grammar, with semantic constraints. It can be specialized into other formalisms, and instantiated into models. A model is an instance of a formalism, that respects the semantic constraints. It cannot be specialized into a formalism, nor instantiated again.

⁸ Thanks to Étienne André, Béatrice Bérard, Fabrice Kordon and anonymous others for their answers.

Strangely enough, answers never cite a reference definition. They all agree on the type/instance difference, and on semantic constraints, but only one among them has added the “specialization” sentence. However, it defines an important property: semantic constraints must be general enough to be applied to any sub formalism, or used by generic algorithms. For instance counting the number of nodes or detecting cycles in graphs can also be applied to automata or Petri nets as they are specific kinds of graphs.

Let us observe a usage example of modular formalisms: representing the state space of a particular instance of the Philosophers problem, and computing its place bounds. The model is represented as a Place/Transition net. We do not use the Petri net of the Model Checking Contest [18], because its modeling does not show some characteristics of our approach. Instead, we use a version where the number of places and transitions depend on the parameter, as shown in [20].

Let us distinguish the various models and formalisms used in this example. They are depicted in Figure 3. “Petri net philosophers” is a formalism for all instances of the Philosophers problem modeled using the chosen representation. “Petri net state space” is a formalism for state spaces of Petri nets. The representation uses the following notations:

- double-lined boxes are formalisms;
- single-lined ones are models;
- inheritance arrows \triangleleft — mean that a formalism or model extends or specializes another one;
- instance arrows \longrightarrow mean that a model is an instance of a formalism;
- reference arrows \dashrightarrow mean that a formalism makes reference to another one, or to a model, they are similar to UML “association arrows”.

Dashed formalisms are usually not considered as formalisms. But given the informal definition of formalisms, they clearly belong to this notion: they describe subclasses of models described by their parent formalism.

Two parts of Figure 3 are of particular interest. First, the “with Bounds” model is just the same as its ancestor, with bound annotations. It is a case where the “Petri net 5 philosophers” model is also a description for the class of models “... with bounds”. Second, the reference arrow between formalism “Petri net 5 philosophers state space” and the model “Petri net 5 philosophers” means that the formalism is defined using a model as parameter. These two examples show that it is difficult to make a clear and uncrossable boundary between models and formalisms. Instead, they seem to be part of a same continuum.

Rahan used only trees and did not distinguish between conifers and hardwood trees. We will follow his path by using layered data to represent both formalisms and models. As there is no intrinsic difference between a formalism and a model, we can thus reduce the notions to the ones of layered data: models (trees, or any boxes in Figure 3), inheritance (lianas, or inheritance and instance arrows in Figure 3) and references (reference signs, or reference arrows in Figure 3).

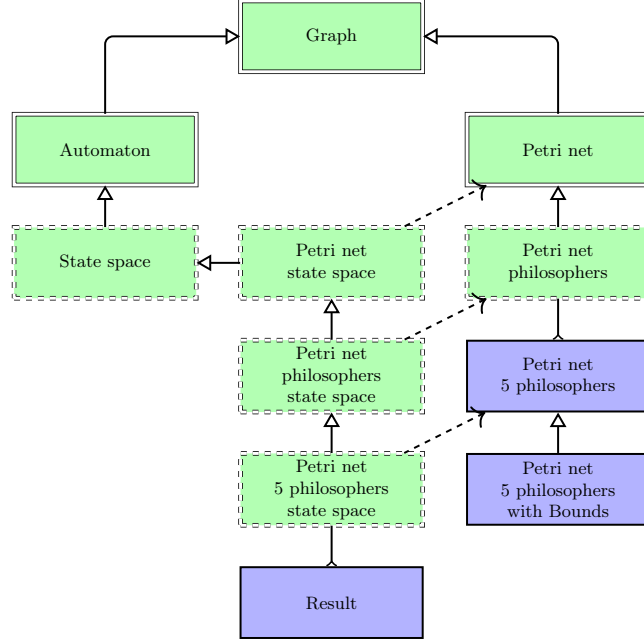


Fig. 1. A hierarchy of formalisms and models

3.1 Consistency

As in class-based systems, models must ensure properties defined by their parent(s). We denote this constraint as the $\models: \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{B}$ relation (\mathbb{P} being a set of formalisms and models, and \mathbb{B} the set of Boolean values). Its first operand is the “formalism”, whereas the second one is the “model”.

Contrary to class-based systems, the \models relation must be transitive in modular formalisms, *i.e.*, $a \models b \wedge b \models c \implies a \models c$, as we would like any graph-based formalism to ensure graph constraints, or all Petri nets variants to ensure Petri nets constraints.

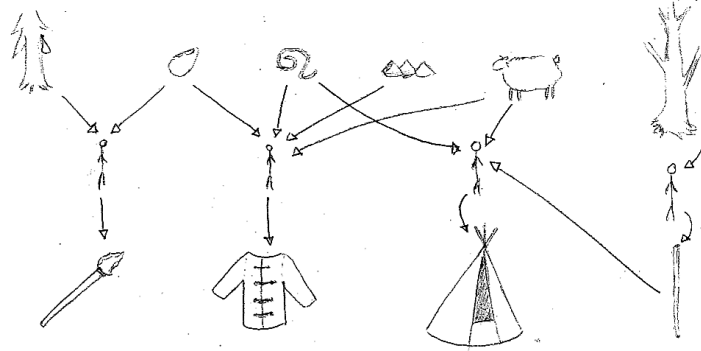
Layered data are only a representation for data; they do not provide a way to ensure the transitivity of \models , as one can redefine almost anything anywhere. For instance, a formalism deriving from the Petri nets could remove the bipartite graph constraints set by the Petri net formalism. Future works should study how to guarantee the transitivity of \models . Our current intuition is that forbidding to redefine an existing value could be sufficient in most of the cases.

4 A Full Hierarchy of Formalisms

In order to show the power of layered data, we present in this section a full hierarchy of formalisms. The chosen formalisms are taken from another problem that Rahan and the villagers encountered.

In the beginning of winter, the villagers have stored wood, animal pelts, and other resources, and they would like to craft their needs for the next year. Their problem is that they cannot allow each craftsperson to use everything they would like, as this behavior can cause resource exhaustion and unnecessary crafts.

Rahan proposes them to draw a representation of the attics and the craftsperson, to allow villagers to discuss resource usage. Each attic should be filled with stones to count available resources. The result is depicted in the picture below⁹.



Villagers discuss a long time, playing with the stones. But they cannot decide what to craft, because they always think about a more optimal resource usage. They end up trying all possible combinations¹⁰, choose the most interesting one, and start crafting for the winter.

We will now take inspiration from Rahan and the villagers, and define the full hierarchy of formalisms they needed, starting from scratch and ending with the state space of a particular Petri net. We will use the previously defined layered data to define everything.

Note that layered data are closer to prototype inheritance than to class-based inheritance: types and instances are not differentiated. Creating a subtype or instantiating a type use the same operation: refining. Types can define mandatory fields, but instances are free to add any other ones.

All steps are presented in this section, using a YAML [8] syntax for readability. The real implementation is Lua [5] code that looks very similar, and can be browsed at <https://github.com/CosyVerif/formalisms>. One part of the definition has been stripped down: the definition of “active” constraints, implemented as Lua functions.

4.1 YAML

YAML describes data structured as trees using mappings, sequences and scalars. Structure is shown through indentation (one or more spaces). Sequence items are denoted by a dash, and mapping keys and values are separated by a colon.

```
key:
  - item 1
  - subkey: value
```

⁹ Surprise! Rahan invented Petri nets a few thousand years before Carl Adam Petri.

¹⁰ Surprise again! These villagers have discovered state space exploration.

4.2 Usability

For better usability, we define two special branch labels: `meta` and `implicit`s. The `meta` branch label is used to distinguish metadata, put within `meta`, from instance data, put outside. The `implicit`s branch states that all direct children of the object, that are neither `meta` nor `refines`, implicitly refine the elements within `implicit`s. These two labels are not mandatory for layered data, but are together a useful feature when defining formalisms and models. We do not add them to the formal definition for readability.

4.3 Collection

The `collection` formalism represents a collection (list) of similar objects. It defines a `type` to optionally specify the type of the collection values; a `container` to optionally specify where these values must be stored (in this case, they will be references); and an optional `minimum` and `maximum` number of elements within the collection.

```
collection:
  meta:
    collection:
      type:      null
      container: null
      minimum:   null
      maximum:   null
  implicit: collection->meta.collection.type
```

By default, `type`, `container`, `minimum` and `maximum` are set to `null`, equivalent to our \perp (nothing) constant. All elements in a collection automatically refine its `type`, because of the `implicit`s key. This feature is not mandatory, but useful when creating instances, as we show at the end of this section. The `collection->meta.collection.type` item represents a reference to the fields `meta.collection.type` of the object that refines the `collection` formalism. It is not YAML syntax, but a convenient representation for us.

Active constraints, implemented as functions, are not represented here. They are in fact part of the formalism and check that the collection constraints are applied on its instances.

4.4 Record

A `record` is a mapping from keys to optionally typed values. It allows specifying that the value associated to a particular key must be of a type, and/or a reference to an element within another object, usually a collection. All formalisms refining `record` should fill the contents of `?` for their own keys to ensure a correct typing of their associated value.

```
record:
  meta:
    ?:
      type:      null
      container: null
```

Active constraints, implemented as functions, are not represented here, but are the main part of the `record` formalism.

4.5 Hyper Multi Graph

The root of all graphs is the formalism for hypermultigraphs, *i.e.*, graphs where an edge can relate more than two vertices (hypergraph [21]), and where there can be several edges between the same vertices (multigraph [7]). This definition is quite big, so we split it in several commented parts.

A **hypermultigraph** defines a **vertex_type**, that is simply a **record** with no defined fields, meaning that there is no mandatory field within a vertex. It also defines a collection of vertices as an instance within the **hypermultigraph**. The **vertex_type** is only a type, and thus defined within the **meta** key, whereas the **vertices** instance is defined outside.

```
hypermultigraph:
  meta:
    vertex_type:
      meta:
        refines: record
  vertices:
    refines: collection
    meta:
      collection:
        type: hypermultigraph->meta.vertex_type
```

Note that the **refines**: key expects a sequence of elements to refine, but that we skip the sequence syntax when there is only one.

In the analogy with trees and lianas presented before, each indentation level is a fork in the tree, reference to other formalisms, such as **record** or **collection** are lianas, labels such as **collection**: or **meta**: are labels painted on branches, and **hypermultigraph->meta.vertex_type** is a reference sign. The *climb* function is used to access nested fields.

The formalism defines in a similar way an **edge_type**, and an **edges** collection. The **edge_type** is more complex, as it contains itself the definition of a type for its arrows, **arrow_type** and a collection of **arrows** as an instance in every edge.

```
hypermultigraph:
  meta:
    edge_type:
      meta:
        refines: record
        arrow_type: ...
      arrows:
        refines: collection
      meta:
        collection:
          type: edge_type->meta.arrow_type
  edges:
    refines: collection
    meta:
```

```

collection:
  type: hypermultigraph->meta.edge_type

```

Each arrow contains a `vertex` field, that must be a reference to a vertex within the `vertices` collection of the graph. As `vertices` is a container of `vertex_type`, the formalism ensures that all edges point to vertices, and that these vertices belong to the same graph. Putting `arrow_type` within `edge_type`, instead of `hypermultigraph` allows to later override the arrow types for each edge instance, or for subtypes of edges.

```

hypermultigraph:
  meta:
    edge_type:
      meta:
        arrow_type:
          refines: record
          meta:
            record:
              vertex:
                container: hypermultigraph->vertices

```

4.6 Graph

We can now define the usual `graphs` above the `hypermultigraphs`. It is done by specifying an exact size for edge `arrows`: 2, one for the source of the edge, one for its target. Moreover, for usability, we specify that an edge has the `source` and `target` attributes. The `arrows` collection contains two predefined arrows, also named `source` and `target` which `vertex` attribute automatically refers to `source` and `target` attributes of the edge.

We cannot specify in the syntax shown in this article that there must not be two edges between the same vertices, *i.e.*, remove the “multigraph” part. In the implementation of layered data and formalisms, such a constraint can be defined as an additional checking function.

```

graph:
  refines: hypermultigraph
  edge_type:
    meta:
      record:
        source:
          container: graph->vertices
        target:
          container: graph->vertices
  arrows:
    meta:
      collection:
        minimum: 2
        maximum: 2
    source:
      vertex: edge_type->source
    target:
      vertex: edge_type->target

```

4.7 Automaton Scheme

An automaton is a `graph` where `vertices` are called `states` and `edges` are called `transitions`, and where states can be set as initial and/or final. The `automaton-scheme` formalism defines the general structure of automata.

Transitions are labeled with letters taken from an **alphabet**, defined within the automaton. The **alphabet** is a collection of *things*, that are not defined in this formalism, but must be specialized in more concrete formalisms.

```
automaton-scheme:
  refines: graph
  alphabet:
    refines: collection
```

The notion of **state_type** (**transition_type**) refines **vertex_type** (respectively **edge_type**), that have been imported by **refines: graph**. The only changes are the new names, and two attributes to states (**initial** and **final**).

```
automaton-scheme:
  meta:
    state_type:
      refines: automaton-scheme->meta.vertex_type
      meta:
        record:
          initial:
            type: boolean
          final:
            type: boolean
    transition_type:
      refines: automaton-scheme->meta.edge_type
      meta: record:
        letter:
          type: automaton-scheme->alphabet.meta.collection.type
          container: automaton-scheme->alphabet
```

The formalism also defines two containers, one for **states**, and one for **transitions**, in the same way the graph defined containers for **vertices** and **edges**.

```
automaton-scheme:
  states:
    refines: collection
    meta:
      collection:
        type: automaton-scheme->meta.state_type
  transitions:
    refines: collection
    meta:
      collection:
        type: automaton-scheme->meta.transition_type
```

The last step is to create a link between the **states** and the **vertices**, and between the **transitions** and the **edges**. The main idea is that, as an automaton is also a graph, all states should be also vertices, and all transitions should also be edges. The following **refines** states that when looking for elements in **vertices**, one must also look at **states**.

```
automaton-scheme:
  vertices:
```

```

    refines: automaton-scheme->states
edges:
    refines: automaton-scheme->transitions

```

4.8 Petri net Scheme

Petri nets are bipartite graphs, with two vertex types: `places` and `transitions`. Edges are called `arcs`. The `petrinet-scheme` formalism defines the general structure of a Petri net, not the Place/Transition net formalism, that will be defined later. It is quite similar to `automaton-scheme`.

Place and transition type both refine the `vertex_type`, imported by `refines: graph`. Places have a `marking` attribute, and arcs a `tokens` one. Their type is not defined in this formalism, as they vary in Petri nets variants. Pre and post arcs refine arcs by specifying the bipartite graph constraint.

```

petrinet-scheme:
  refines: graph
  meta:
    place_type:
      refines: petrinet-scheme->meta.vertex_type
      meta:
        record:
          marking: null
    transition_type:
      refines: petrinet-scheme->meta.vertex_type
    arc_type:
      refines: petrinet-scheme->meta.edge_type
      meta:
        record:
          tokens: null
    pre_arc_type:
      refines: petrinet-scheme->meta.arc_type
      meta:
        record:
          source:
            container: petrinet-scheme->places
          target:
            container: petrinet-scheme->transitions
    post_arc_type:
      refines: petrinet-scheme->meta.arc_type
      meta:
        record:
          source:
            container: petrinet-scheme->transitions
          target:
            container: petrinet-scheme->places

```

`petrinet-scheme` defines containers for `places`, `transitions` and `arcs`. It also sets the link between them and the `vertices` and `edges` container of `graph`.

This is similar to what exists in `automaton-scheme`. The previous definition of `petrinet-scheme` is completed as below (the first line is repeated only for readability).

```
petrinet-scheme:
  places:
    refines: collection
    meta: collection:
      type: petrinet-scheme->meta.place_type
  transitions:
    refines: collection
    meta: collection:
      type: petrinet-scheme->meta.transition_type
  pre_arcs:
    refines: collection
    meta: collection:
      type: petrinet-scheme->meta.pre_arc_type
  post_arcs:
    refines: collection
    meta: collection:
      type: petrinet-scheme->meta.post_arc_type
  arcs:
    refines:
      - petrinet-scheme->pre_arcs
      - petrinet-scheme->post_arcs
  vertices:
    refines:
      - petrinet-scheme->places
      - petrinet-scheme->transitions
  edges:
    refines: petrinet-scheme->arcs
```

4.9 State space of Petri net

The state space of a Petri net is simply an automaton where states are labeled with petri net states, and transitions are labeled with petri net bindings. The notions of states and bindings is not defined in the `petrinet` formalism, so we first define them in a `petrinet-behavior` formalism. Contrary to the ones we already defined, `petrinet-behavior` does not refine `graph` or `petrinet`. It takes the `petrinet` as a parameter instead. This formalism must be refined for each Petri net variant.

```
petrinet-behavior:
  refines: record
  meta:
    record:
      petrinet:
        type: petrinet-scheme
```

The `state_type` is set as refining the Petri net (markings are overloaded in each state), whereas the `binding_type` has the fired transition as attribute.

```

petrinet-behavior:
  meta:
    state_type:
      refines: petrinet-behavior->petrinet
    binding_type:
      refines: record
    meta:
      record:
        transition:
          container: petrinet-behavior->petrinet.transitions

```

A Petri net state space formalism is built upon a Petri net, and a Petri net behavior. The state space takes as parameter the Petri net, and passes it to a `petrinet-behavior` instance.

```

petrinet-statespace:
  refines: automaton-scheme
  meta:
    record:
      petrinet: petrinet-scheme
  behavior:
    refines: petrinet-behavior
    petrinet: petrinet-statespace->petrinet

```

Automaton states are states of the behavior, while the transition alphabet uses the Petri net behavior bindings.

```

petrinet-statespace:
  meta:
    state_type:
      refines: petrinet-statespace->behavior.meta.state_type
    alphabet:
      meta:
        collection:
          type: petrinet-statespace->behavior.meta.binding_type

```

4.10 P/T net:

The Place/Transition net formalism is a Petri net where place markings are natural numbers, and arcs are also valued by positive natural numbers.

This formalism sets the types of place markings and arc tokens to `natural`, that is a predefined formalism for natural integer literals. The `petrinet-pt` formalism also sets default values for place markings and arc values.

```

petrinet-pt:
  refines: petrinet-scheme
  meta:
    place_type:
      marking: 0
    meta:
      record:
        marking: natural
    arc_type:
      tokens: 1
    meta:
      record:
        tokens: positive

```

We can also create the formalism for their behavior. It simply refines the `petrinet-behavior` one with no changes, as bindings are composed of the fired transition only.

```
petrinet-behavior-pt:
  refines: petrinet-behavior
  record:
    petrinet:
      type: petrinet-pt
```

4.11 P/T net for Philosophers:

We can continue to specialize Place/Transition nets to define the formalism for Place/Transition Philosophers Petri nets. It defines a different place type for each “kind” of place (`think, ...`), and does the same for transitions.

```
petrinet-philosophers:
  refines: petrinet-pt
  meta:
    place_type:
      record:
        identifier: natural
    transition_type:
      record:
        identifier: natural
    think:
      refines: petrinet-philosophers->meta.place_type
    fork:
      refines: petrinet-philosophers->meta.place_type
    ...
    release:
      refines: petrinet-philosophers->meta.transition_type
    ...
```

This formalism can be useful for graphical representations of instances: positions can be specified in polar coordinates for each place or transition type, depending on the philosopher identifier.

4.12 Philosophers model

The final model is the Philosophers instance. Each place or transition defines its philosopher identifier and refines its place or transition type. Note that there is here no need to specify the `refines` for each place, transition or arc, as the `implicits` of `collection` define implicitly the refinements.

```
philo:
  refines: petrinet-philosophers
  places:
    think1:
      refines: philo->meta.think
      identifier: 1
    ...
```

```

transitions:
  release1:
    refines: philo->meta.release1
    identifier: 1
  ...
arcs:
  arc1:
    source: philo->places.think1
    target: philo->transitions.take-left1
  ...

```

The real implementation of the formalisms and models is as data represented in the Lua [5] programming language. It is thus easy to write a function that generates the Philosophers model from a positive number parameter.

5 Comparison with existing solutions

We can distinguish two approaches for formalism or language definition: the formal methods and the software engineering ones. In the formal methods domain, formalisms usually are a mathematical definition of a class of models with their semantics, seldom derived from an already existing class of models. In the software engineering domain, the definition is composed of the following parts, but unoften uses the mathematical definition:

- an abstract syntax of the concepts and their relationships, from a syntactical point of view, usually described using a metamodel, *e.g.*, in UML class diagram notation;
- a concrete syntax defining the notation that will be used to represent actual models of the formalism; it can make use of any sense available to the human programmer or the computers, but is usually textual or graphical; it can be intended either for human use, *e.g.*, graphical notations, or for computer use, *e.g.*, interchange formats;
- a syntactic mapping between the abstract syntax and the concrete one, that allows the rigorous translation of the concrete syntax elements into the abstract syntax ones, thus enabling programs or human beings to figure out the models being described, and interpret them properly.

Recent works, such as the ISO/IEC 15909 Standard for Petri nets [14], [15] merge the two approaches by adding a semantic mapping between the formal definition and the abstract syntax, that allows the rigorous interpretation of any model instance represented in the abstract syntax using the semantics provided with the formal definition.

This standard sets the ground work for Petri net formalisms definition by providing formal definitions and semantics, abstract syntax using UML class diagrams, semantic mappings, and a transfer format (PNML) are provided for three types of Petri nets: P/T nets, Symmetric nets and High-level Petri nets. It is implemented in the PNML Framework [13]. Recent works [12] have targeted

the modularity of these definitions. Several tools implement more or less this approach, for instance the ePNK [17] platform dedicated to Petri net tools, and the CosyVerif [1] platform that is more formalism agnostic.

6 Conclusion

This article defines a data structure, called *layered data*. It stores data as layers of directed rooted trees, and allows accessing them in a flattened view. With only a few simple constructs (refinement, labels and references), layered data allow building abstract syntaxes for complex modular formalisms and models.

As an example, we have used layered data to build several formalisms, from the hypermultigraphs to Petri net instances and the state space of P/T nets. All these definitions remain both modular and compact. The semantics usually associated to formalisms is not encoded in layered data, that only focus on the abstract syntaxes. Future works should consider adding semantic information within the formalisms. Our approach should allow to define hierarchies for both formalisms and semantics, in a synchronized way.

Layered data [4] and the hierarchy of formalisms [2] presented in this article are hosted on GitHub [3] and implemented in Lua [5]. This language is intended for embedding into host languages, and allows porting layered data to any language (for instance C++, Java or even JavaScript) by using only existing binding libraries or generators.

Formalisms and models shown in this article are all defined within the scope of the CosyVerif [1] project: a platform dedicated to the verification of complex systems. The whole platform is open source, released under the permissive MIT software license [6]. In this platform, formalisms are defined by researchers, who also create tools for analysis of their models; whereas engineers or students create models and use tools on them.

We also show in this article that the distinction between formalisms and models is not as simple as it seems to be. The formal methods community lacks a clear definition for these two notions. We suggest that formalisms do not exist on their own, but that any model used to define another one should be called a formalism **of** its model.

References

1. CosyVerif: a platform dedicated to the verification of complex systems, <http://cosyverif.org>
2. Cosyverif formalisms, <https://github.com/CosyVerif/formalisms>
3. GitHub, <https://github.com>
4. Layered Data, <https://github.com/saucisson/lua-layereddata>
5. Lua Language, <http://lua.org>
6. Mit license, <https://opensource.org/licenses/MIT>
7. Multigraph, <http://www.encyclopediaofmath.org/index.php?title=Multigraph&oldid=13089>

8. YAML Ain't Markup Language, <http://www.yaml.org>
9. Étienne André, Barbot, B., Demoulins, C., Hillah, L.M., Hulin-Hubard, F., Kordon, F., Linard, A., Petrucci, L.: A modular approach for reusing formalisms in verification tools of concurrent systems. In: Groves, L., Sun, J. (eds.) *Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings*. Lecture Notes in Computer Science, vol. 8144, pp. 199–214. Springer (2013), http://dx.doi.org/10.1007/978-3-642-41202-8_14
10. Étienne André, Lembachar, Y., Petrucci, L., Hulin-Hubard, F., Linard, A., Hillah, L., Kordon, F.: Cosyverif: An open source extensible verification environment. In: *2013 18th International Conference on Engineering of Complex Computer Systems*, Singapore, July 17-19, 2013. pp. 33–36. IEEE Computer Society (2013), <http://dx.doi.org/10.1109/ICECCS.2013.15>
11. Diestel, R.: *Graph theory*. Graduate texts in mathematics, Springer, Berlin (2005), <http://opac.inria.fr/record=b1102131>, cop. 2006 pour l'édition brochée
12. Hillah, L., Kordon, F., Lakos, C., Petrucci, L.: Extending pnml scope: A framework to combine petri nets types. In: *T. Petri Nets and Other Models of Concurrency [16]*, pp. 46–70, http://dx.doi.org/10.1007/978-3-642-35179-2_3
13. Hillah, L., Kordon, F., Petrucci, L., Trèves, N.: PNML framework: An extendable reference implementation of the petri net markup language. In: Lilius, J., Penczek, W. (eds.) *Applications and Theory of Petri Nets, 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6128, pp. 318–327. Springer (2010), http://dx.doi.org/10.1007/978-3-642-13675-7_20
14. ISO/IEC: *Software and Systems Engineering - High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation*, International Standard ISO/IEC 15909 (December 2004)
15. ISO/IEC: *Software and Systems Engineering - High-level Petri Nets, Part 2: Transfer Format*, International Standard ISO/IEC 15909 (February 2011)
16. Jensen, K., van der Aalst, W.M.P., Marsan, M.A., Franceschinis, G., Kleijn, J., Kristensen, L.M. (eds.): *Transactions on Petri Nets and Other Models of Concurrency VI*, Lecture Notes in Computer Science, vol. 7400. Springer (2012), <http://dx.doi.org/10.1007/978-3-642-35179-2>
17. Kindler, E.: The epnk: An extensible petri net tool for PNML. In: Kristensen, L.M., Petrucci, L. (eds.) *Applications and Theory of Petri Nets - 32nd International Conference, PETRI NETS 2011, Newcastle, UK, June 20-24, 2011. Proceedings*. Lecture Notes in Computer Science, vol. 6709, pp. 318–327. Springer (2011), http://dx.doi.org/10.1007/978-3-642-21834-7_18
18. Kordon, F., Linard, A., Buchs, D., Colange, M., Evangelista, S., Lampka, K., Lohmann, N., Paviot-Adet, E., Thierry-Mieg, Y., Wimmel, H.: Report on the model checking contest at petri nets 2011. In: *T. Petri Nets and Other Models of Concurrency [16]*, pp. 169–196, http://dx.doi.org/10.1007/978-3-642-35179-2_8
19. Lecureux, R., Chéret, A.: Rahan. Lecureux Productions
20. Rp22: A petri net model of the dining philosophers problem, with 4 philosophers (2008), <https://upload.wikimedia.org/wikipedia/commons/7/78/4-philosophers.gif>
21. Sapozhenko, A.: Hypergraph, <http://www.encyclopediaofmath.org/index.php?title=Hypergraph&oldid=11526>