# HyMeKo Language: Describing Complex Hypergraph-Like Data

**Csaba Hajdu[1,2], Adam B. Csapo[3,4]**

[1] Department of Informatics
Széchenyi István University
Egyetem tér 1, Győr, 9026, Hungary
Email: hajdu.csaba@ga.sze.hu

[2] Székesfehérvár Science and Innovation Park
Székesfehérvár, 8000, Hungary

[3] Corvinus Institute for Advanced Studies /
Institute of Data Analytics and Information Systems
Corvinus University of Budapest
Fővám tér 8, Budapest, 1093, Hungary
Email: adambalazs.csapo@uni-corvinus.hu

[4] Hungarian Research Network
Piarista u. 4, Budapest, 1052, Hungary

*Abstract: Numerous applications of computer science — including artificial intelligence, robotics, and cyber-physical systems — rely on highly connected data structures. Such complex information is most naturally and compactly described using a hypergraph-based approach, which enables concise representation of many-to-many relationships. In this paper, we introduce HyMeKo, a formal markup language designed to represent highly connected data based on hypergraph theory. Unlike traditional formats such as XML or JSON, HyMeKo offers a significantly more concise and semantically expressive way to model complex relationships by organizing data into a hypertree-based structure. The language supports template-based modeling and inheritance, enabling reusable, modular, and scalable data descriptions. HyMeKo is implemented as an LALR(1)-compliant language, allowing efficient parsing and transformation of structured data into hypergraphs. We provide a formal definition of the language, its supported operations, and relational rules, along with a comparative analysis demonstrating its syntactic efficiency. Application examples include robotic system descriptions, neural network architectures, and structured LLM prompts. We further present a structural complexity analysis showing that HyMeKo achieves a (k+1)-fold reduction in representational overhead compared to RDF reification for k-ary relationships, and provide an explicit comparison with RDF, OWL, and GraphQL. Reference implementations exist in both Python (PyLark) and Rust (LALRPOP).*

*Keywords: hypergraphs; semantic description; data description language; markup language; domain modeling*

# 1 Introduction

Complex systems in robotics, cyber-physical systems, and AI rely on structured knowledge representations that capture many-to-many relationships. Existing description formats — XML-based (URDF, SDF, OWL, RDF), JSON, YAML — are fundamentally constrained to binary relationships. Expressing n-ary connections explicitly in binary-edge schemas requires introducing auxiliary junction nodes (reification), which artificially inflates the graph structure and decreases semantic flexibility. Furthermore, these formats lack native support for directionality semantics and structural reuse through inheritance.

This article introduces HyMeKo (Hypergraph Model Cognition Framework), a formal markup language designed to represent highly connected data using hypergraph theory. Unlike traditional formats, HyMeKo organizes data into a hypertree-based structure where hyperedges natively express n-ary, directed relationships with signed incidence semantics. The language supports template-based modeling and inheritance, enabling modular and scalable descriptions.

## Contributions

The contributions of this paper are as follows: (1) a formal LALR(1)-compliant language definition with EBNF grammar; (2) algebraic properties of the language's relational operations (import, stereotype, reference); (3) a comparative size analysis against XML, JSON, and YAML; (4) application examples in robotic description, neural network architecture, and structured LLM prompting. A reference parser implementation exists in Python (PyLark); (5) a structural comparison with RDF, OWL, and GraphQL, quantifying the reification overhead inherent in binary-edge representations; and (6) an evaluation across multiple benchmark models using structural and size metrics beyond character count.

Furthermore a high-performance Rust implementation using LALRPOP (for the LALR(1) parsing) has since been completed.

# 2 Related work

Before detailing the related work, we note that HyMeKo targets LALR(1) compatibility [1], enabling parsing in $O(n)$ steps.

## 2.1 Semantic Web and Graph Query Standards

Current semantic web standards, such as RDF/Turtle and OWL are predominantly based on XML [2] or JSON-based schemas. While widespread, these standards introduce numerous problems for each format:

- **RDF/Turtle**: use a binary triple model to describe semantic information. Utilizing N-ary information requires reification [3]. Overhead numbers have been identified by numerous authors[4, 5].
- **OWL**: inherits RDF's binary limitation that can be overcome with "class-as-relation" pattern (W3C recommendation). However, authors like Salguero [6] show the error-proneness and domain pollution of this method.
- **GraphQL**: while query language, many-to-many relations require junction types with no incidence semantics.

Three structural limitations emerge across all surveyed standards, which HyMeKo - the proposed language - is designed to address:

1. Relationships are fundamentally constrained to binary edges, requiring auxiliary constructs for n-ary connections
2. Signed incidence semantics, distinguishing the role and directionality of participants within a single relationship, have no native representation
3. Structural inheritance and template-based reuse are either absent or delegated to external tooling

## 2.2 Description formats

Beyond the semantic web standards discussed in subsection 2.1, domain-specific formats inherit the same XML verbosity while introducing additional structural constraints. In robotic simulation, URDF (used in Gazebo [7]) restricts models to acyclic trees with no support for kinematic loops [8], and XML macro tools (Xacro) introduce additional points of failure rather than solving the underlying problem [9]. SDF allows kinematic loops but with unintuitive syntax [2]. In virtual environments, WOM uses a React XML composition tree with parent-child relationships, also making kinematic loops unintuitive. Webots [10], based on VRML, offers more flexible element referencing among the surveyed formats. Table 1 summarizes these properties.

| Name | Schema format | Component relationship | Kinematic loops allowed |
|---|---|---|---|
| URDF | XML | Linear unfolded | Prohibited |
| SDF | XML | Linear unfolded | Allowed |
| WOM | React XML | Parent-child | Unavailable |
| Webots | JSON/XML | Parent-child | Allowed |

Table 1: Comparison of some of the description formats used in the description of simulators and virtual reality systems

## 2.3 Hypergraphs

In graph theory, a hypergraph $H = (V, E = (e_i)_{i \in I})$ generalizes a graph $G(V, E_x)$ by allowing each edge $e_i \subseteq V$ to connect an arbitrary number of vertices simultaneously, rather than exactly two. The index set $I$ is finite, and the elements of $E$ are called hyperedges. The following definitions follow Bretto [11] and Dai and Gao [12]; only those properties referenced later in the paper are listed:

- *Order of the hypergraph*: cardinality of vertices $V$, $n := |V|$
- *Size of the hypergraph*: cardinality of edges $E$, $m := |E|$
- *Empty hypergraph*: $V = \varnothing$, $E = \varnothing$
- *Trivial hypergraph*: $V \neq \varnothing$, $E = \varnothing$
- *Loop*: a $e \in E$ hyperedge is a loop, if $|e| = 1$.
- *Isolated vertex*: an isolated vertex in a hypergraph $H(V, E)$ is any $x \in V \setminus \bigcup_{i \in I} e_i$. Conversely, a hypergraph has no isolated vertex if $\bigcup_{i \in I} e_i = V$
- *Induced subhypergraph of* $H(V, E)$: given $V' \subseteq V$, the induced subhypergraph is $H(V') = (V', E')$ where:

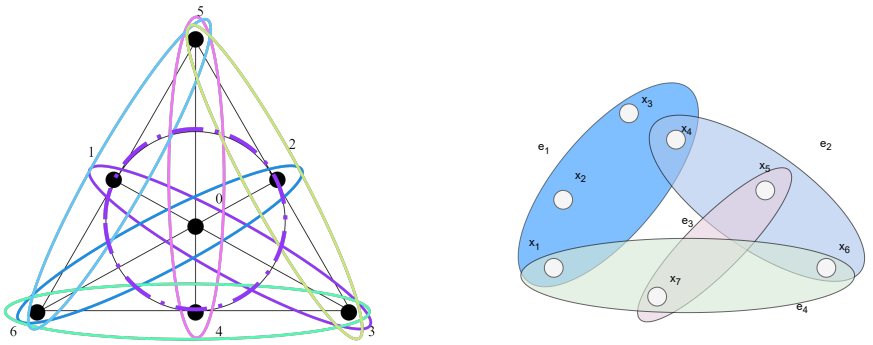
  $$E' = \{e_i \in E : V(e_i) \cap V' \neq \varnothing \wedge (|V(e_i) \cap V'| \geq 2 \vee \text{loop}(e_i))\}$$

  Note that $E'$ can be represented as a multi-set.
- *Subhypergraph of* $H(V, E)$: given $V' \subseteq V$, and $J \subset I$ $H'$ subhypergraph is the hypergraph $H = (V', E' = (e_j)_{j \in J} | \forall e_j \in E' : e_j \subseteq V')$
- *Partial hypergraph of* $H(V, E)$: generated by $J \subseteq I$, $H' = (V', (e_j)_{j \in J})$, where $\bigcup_{j \in J} e_j \subseteq V'$ ($V' = V$ is possible).
- *Star in* $H(x)$: centered in $x \in V$, the set of hyperedges $(e_j)_{j \in J}$ containing $x$.

- *Degree*: the degree of $x \in V$, a loop containing $x$, given a star $d(x) = |J|$. Without repeated hyperedges, the degree is denoted as $d(x) = |H(x)|$. The maximal degree of a hypergraph $H$ is denoted by $\Delta(H)$.
- *Regularity*: if each vertex has the same degree, the hypergraph is regular, or *k-regular* $\forall x \in V, d(x) = k$.
- *Rank and co-rank*: the maximum cardinality of a hyperedge in the hypergraph: $r(H) = max_{i \in I}|e_i|$, the minimum cardinality of a hyperedge is the co-rank $cr(H) = min_{i \in I}|e_i|$. If $r(H) = cr(H) = k$ the hypergraph k-uniform.

A *simple hypergraph* is a hypergraph $H(V; E)$ such that $e_i \subseteq e_j \Longrightarrow = j$ and has no repeated hyperedge. A hypergraph is *linear* if it is simple, and $\forall i, \forall j (i, j : |e_i \cap e_j| \leq 1 \wedge i, j \in I \wedge i \neq j)$.

A well-known example of a hypergraph is the Fano plane, depicted in Figure 1a. It is defined by seven hypervertices $V := \{0, 1, 2, \ldots, 6\}$ and seven hyperedges $E := \{(0; 1; 3), (0; 4; 5), (0; 2; 6), (1; 2; 4), (1; 5; 6), (2; 3; 5), (3; 4; 6)\}$, forming a 3-uniform, 3-regular hypergraph. The Fano plane is also commonly represented as a multi-set, as shown in Figure 1b.




(a) Fano graph (Fano-plane)

(b) Hypergraph as multi-set

Figure 1: Typical illustrative example of hypergraphs

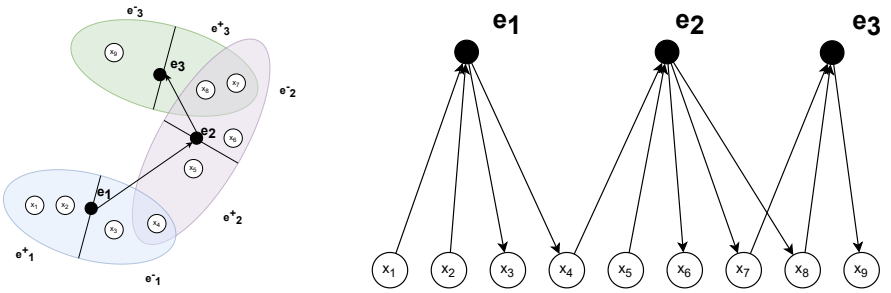
## 2.4 Directed hypergraphs (dirhypergraphs)

Directionality can be used with hypergraphs as well, similar to simple graphs. A *directed hypergraph (dirhypergraph)* is an ordered pair of $\vec{H} = \left(V; \vec{E} = \{\vec{e_i} : i \in I\}\right)$, where $V$ is a finite set of vertices and $\vec{E}$ is a set of hyperarcs with finite index set $I$. Each hyperarc $\vec{e_i} \in \vec{E}$ is an ordered pair $e_i = \left(\vec{e_i^+} = (e_i^+, i); \vec{e_i^-} = (i, e_i^-)\right)$,

where hyperedges are segmenting the connected nodes to two distinct categories (i.e., limbs):

- $\vec{e_i^+} \subseteq V$ of vertices of $\vec{e_i^+}$, representing *tail* of the hyperarc. The set of
- $\vec{e_i^-} \subseteq V$ of vertices of $\vec{e_i^-}$, representing the *head* of the hyperarc.
- Ensure that all $\vec{e} = (\vec{e^+}, \vec{e^-}) \in \vec{E}$ and $e^+ \cap e^- = \varnothing, e^+ \neq \varnothing, e^- \neq \varnothing$

An illustrative example of a directional hypergraph is depicted in Figure 2a




(a) Dirhypergraph (b) Dirhypergraph as an incidence graph

Figure 2: Typical illustrative example of dirhypergraphs

# 3 Definition of HyMeKo language

The modeling process with the framework is shown in Figure 3. The metaelements and domain transformation rules are external subject to a specific application area. These meta-definitions can be reused through multiple elements.

## 3.1 Language definition

The HyMeKo language is defined as an LALR(1)-compliant grammar, enabling extension to different application domains. It supports two modes of use:

1. Direct description of structured information,
2. Domain-specific modeling through metaelements: reusable templates imported into a target description and transformed into a processable hypergraph structure.

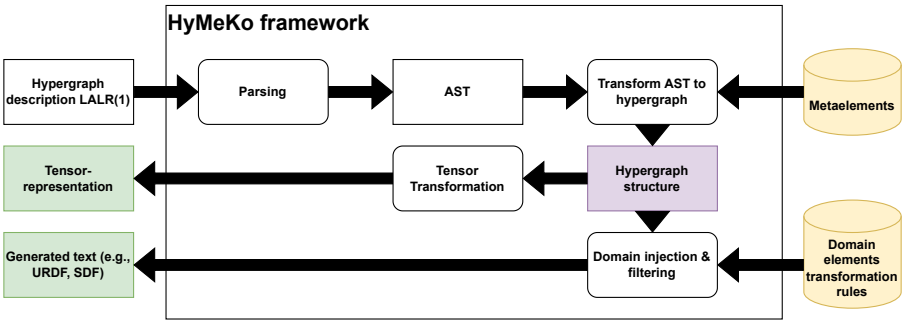
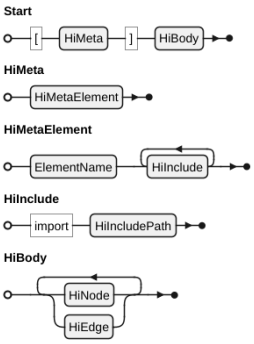
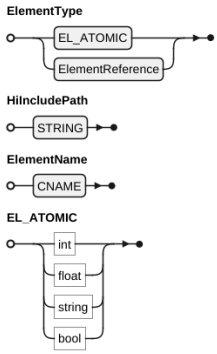



Figure 3: Modeling with the HyMeKo framework

The grammar is organized hierarchically. Figure 4 shows the top-level composite structure and primitive types. Hypervertices (Figure 5a) can be nested recursively, forming a composite tree $T$ for structural organization. Hyperedges (Figure 5b) contain hyperarcs connecting hypervertices and element fields. Element references (Figure 6a) and field definitions (Figure 6b) complete the grammar.



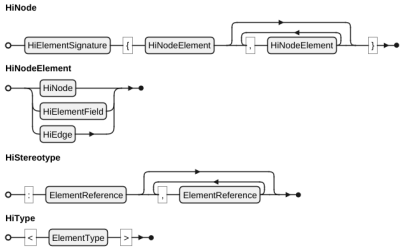

(a) HyMeKo language composite language elements



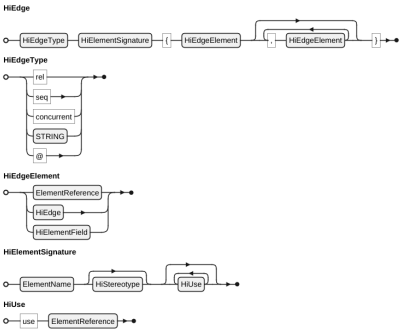

(b) Hymeko Language primitives

Figure 4: HyMeKo language EBNF elements

The grammar reflects three design decisions tied to the hypergraph semantics. The ’’ prefix distinguishes hyperedges from hypervertices at the lexical level, enabling the parser to resolve element type without lookahead. Relation directions (’->’, ’<-’, ’–’, ’<>’) are first-class tokens encoding signed incidence directly in the syntax. Stereotypes (:) and the use keyword provide structural inheritance and domain import at the language level, rather than delegating reuse to external tooling.

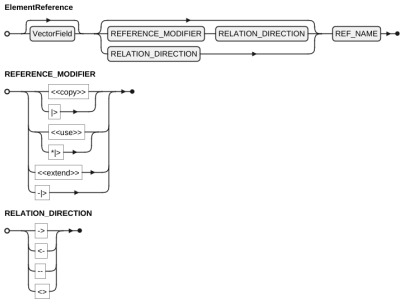



(a) Hypervertex definition in Hymeko Language

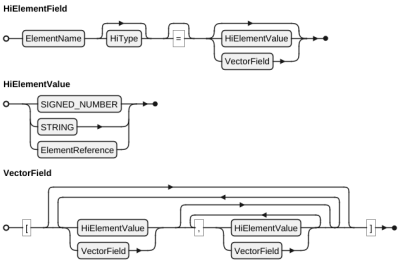



(b) Hyperedge definition in Hymeko Language

Figure 5: Hypergraph element definitions




(a) Element reference definition in Hymeko Language




(b) Element field definition in Hymeko Language

## 3.2 Relationship definitions

### 3.2.1 Extend the domain of elements

The language can relate to other elements with operations and relations (see subsubsection 3.2.2). On the other hand, relationships can be built between elements not initially present in the domain of the hypergraph. Therefore, the extension of the initial domain of the hypergraph is needed. In HyMeKo, it is done with the *"import"* tag, which adds additional hypergraphs into the domain of hypergraphs and, consequently, to the elements.

The relation extends the domain of the initial hypergraph $H_1(V_1, E_1)$ and its elements $D := H_1 = (V_1, E_1)$. So therefore, if a new hypergraph $H_2(V_2, E_2)$, all of its elements will be reachable in the domain $D^* := D \cup H_2 = V_1, E_1, V_2, E_2$. For the remainder of the paper, it is denoted as $\hookleftarrow$. This relation is fundamental

if the domain is to be further extended with new elements, with the following properties on hypergraphs:

- **Reflexivity**: self import of a graph leads to itself $H(V, E)$, so $H \hookleftarrow H = H$. When the domain of elements are investigated, it is easy to see, that $D := H_1 \cup H_1 = H_1$.
- **Transitivity**: transitive imports are present in further imports. If $H_2 \hookleftarrow H_1$, then $H_3 \hookleftarrow H_2$, then $H_3 \hookleftarrow H_1$. Indeed, the domain is extended with the elements $D := H_3 \cup (H_2 \cup H_1) = H_1, H_2, H_3$ and therefore all elements of $H_1, H_2, H_3$.
- **Antisymmetry**: the import relation is antisymmetric, as there is a directionality of this operation, as well as importing hypergraphs typically not inducing any effect on the reference: $H_1, H_2, H_2 \hookleftarrow H_1 \neq H_1 \hookleftarrow H_2$
- **Associativity**: importing can be grouped with parentheses, resulting in the same. Intuitively, union operation is associative, therefore in the case of importing is true: $H_3 \hookleftarrow H_2 \hookleftarrow H_1 = H_3 \hookleftarrow (H_2 \hookleftarrow H_1)$.
- **Closedness**: importing hypergraphs do not induce anything else than other hypergraphs and their respective elements (vertices, edges).
- **Transitive closure**: importing hypergraphs and inducing them in a recursive operation avoids the explosion of domain space (i.e., infinite reference). With union operation regarding domain (and the extension with hypergraphs), this is true.

### 3.2.2 Relations defined between elements

The following section overviews the relationships that can be defined between elements of the hypergraph $H(V, E)$ (i.e. any $v \in V$ or $e \in E$).

If the level of the language definition, would require all type of relationships to be defined using only edges and edge relationships, it would lead to a verbose model, therefore special element-specific relationships are also defined. Given a $H(V, E)$ (or in the extended case on the domain of hypergraphs $D := \bigcup_i H_i$) the following special relationships (relations) are defined element-wise:

- **Stereotypes**: A relationship describing that an element is redefining (is-a type of) another element, that is it uses the elements or copies.
  - *Copying (inheritance)*: the element reuses the stereotyped element as an instance, meaning all elements are copied as subelements. A typical use case would be modeling kinematic skeletons: legs and arms use the same structure description, but they are instanced separately to

allow the setup of animation key points during kinematic detection. For simplicity, the operation is referred as $x \sqsubset y, x, y \in V$ or $x \sqsubset y, x, y \in E$.

- *Stereotype reference (Realization)*: in this case, the elements are not instanced, only referred. Consequently, that the component uses the information, connections and attributes handled by the stereotype. For simplicity, it is denoted as $x \triangleright y, x, y \in V$ and

- **Element reference**: an attribute can refer to another hypergraph element. On the description language level, hyperedge relations are attribute references, referring to other elements.

## 3.3 Formal rules

### 3.3.1 Properties of relations between hypergraph elements

Both the stereotype generalization (copying) operation $x \sqsubset y$ and the stereotype reference (realization) $x \triangleright y$, where $x, y \in V$ or $x, y \in E$, share the following algebraic properties:

- **Reflexivity**: $\forall x \in V \cup E$, $x \sqsubset x \Rightarrow x$ and $x \triangleright x \Rightarrow x$. Both operations apply indifferently to hyperedges and hypervertices.
- **Closedness**: both operations remain in the domain $D$ (minimally $D := H$). New elements can be introduced by extending the domain via import (e.g., $D^* := D \cup H_2$).
- **Transitivity**: consecutive applications compose associatively.
- **Asymmetry**: neither operation induces any change in the referred element.
- **Associativity**: $\forall x, y, z \in V \cup E$, $cp(cp(x, y), z) = cp(x, cp(y, z))$.

Together, these properties establish that both stereotype operations form closed, reflexive, transitive, and asymmetric relations over the elements of the hypergraph.

## 3.4 Parsing

The language is compatible with LALR(1) parsers. Reference implementations exist in Python (PyLark)[1] and Rust (LALRPOP)[2]. This means, that any hypergraph description can be parsed in $O(n)$ steps - therefore validated. The associated

1 `https://github.com/kyberszittya/himeko_lang`
2 `https://github.com/kyberszittya/hymeko_framework_rust`

grammar is depicted in Listing 1. The listing does not include PyLARK specific constructs such as the ignoring of the whitespaces and importing basic elements.

Listing 1: HyMeKo hypergraph description language (LARK)

```
// hi_*: abbreviation for Himeko mandatory language elements
start: "[" hi_meta "]" hi_body
// Metadata section
?hi_meta: hi_metaelement
hi_metaelement: element_name [(hi_include)+]
// Metadata inclusion
hi_include: "import" hi_include_path
hi_include_path: STRING
// Body section
hi_body: (hi_node)+
// Node element
hi_node: hi_element_signature "{" (hi_node_element ((",")? hi_node_element)*)? "}"
hi_node_element: hi_node | hi_element_field | hi_edge
// Edge element
hi_edge: hi_edge_type hi_element_signature "{" (hi_edge_element ( "," hi_edge_element)*)? "}"
hi_edge_type: "rel" | "seq" | "concurrent" | STRING | "@"
hi_edge_element: (element_reference | hi_edge | hi_element_field)
// Element signatures and fields
hi_element_signature: element_name hi_stereotype? (hi_use)*
hi_element_field: element_name hi_type? ("=")? (hi_element_value | vector_field)?
// Stereotypes
hi_stereotype: ":" element_reference ("," element_reference)*
hi_use: "use" element_reference
hi_type: "<" element_type ">"
element_type: EL_ATOMIC | element_reference
// Vector field definition
vector_field: "[" (( (hi_element_value|vector_field) ("," (hi_element_value|vector_field))*) ) "]"
hi_element_value: SIGNED_NUMBER | STRING | element_reference
// Element names and references
element_name: CNAME
element_reference: vector_field? ((REFERENCE_MODIFIER RELATION_DIRECTION)|RELATION_DIRECTION)? REF_NAME
REFERENCE_MODIFIER: "<<copy>>" | "<<use>>" | "<<extend>>"
// Direction for relationships
RELATION_DIRECTION: "->" | "<-" | "--" | "<>"
// Atomic field types
EL_ATOMIC: "int" | "string" | "real"
REF_NAME: /[_a-zA-Z][_a-zA-Z0-9.]*/
```

# 4 Discussion

## 4.1 Syntactic compactness analysis

To illustrate syntactic compactness, we first compare minimal linear node descriptions in HyMeKo, JSON, XML, and YAML under the simplifying assumption of one-character identifiers and a single root element. Representative patterns are shown in Figure 7, while the corresponding character-count formulas are summarized in Table 2.

Under these assumptions, HyMeKo requires $3n$ characters, or $3n + 3$ if the root element is included. JSON requires $6n - 1$ characters, or $6n + 4$ with the root element. XML requires $7n$ characters, or $7(n+1)$ with the root element, while the minimal empty-node case yields $4n + 7$. YAML requires $4n$ characters, or $4n + 2$ with the root element.

We next compare linear hypergraph relation descriptions. Consider the hypergraph shown in Figure 8, consisting of a single hyperedge incident to $r$ hyperarcs. Representative encodings in HyMeKo, XML, YAML, and JSON are shown in Figure 9.

**HyMeKo**

```
n{
a{}
b{}
...
z{}
}
```

**JSON**

```
{
"n": {
  "a":{},
  "b":{},
  ...
  "z":{}
}
}
```

**XML**

```
<n>
<a></a>
<b></b>
...
<z></z>
</n>
```

**YAML**

```
n:
 a: a
 b: a
 ...
 z: a
```

Figure 7: Minimal linear node description patterns in the compared schemas

| Schema | Character-count formula |
|---|---|
| HyMeKo | $3n$ ($3n + 3$ with root) |
| JSON | $6n - 1$ ($6n + 4$ with root) |
| XML | $7n$ ($7(n + 1)$ with root) |
| XML (empty-node case) | $4n + 7$ |
| YAML | $4n$ ($4n + 2$ with root) |

Table 2: Character-count formulas for minimal linear node descriptions

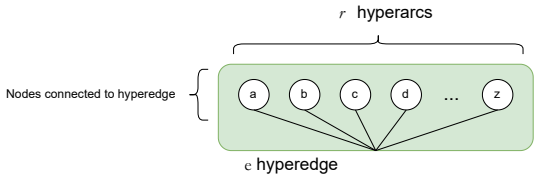



Figure 8: Linear hypergraph used for compactness analysis

In HyMeKo, the structure is represented by a single hyperedge declaration and requires $4r + 4$ characters.

For XML, two representative encodings can be considered. The first uses nested relation elements and yields $r(2n+5)+7$ characters without explicit directionality, and $r(2n + 10) + 7$ when directionality is included. The second uses attributes; for $n = 1$, this results in $4r + 4$ characters without explicit directionality and $8r + 4$ with directionality.

YAML [13] represents hierarchy through indentation. A straightforward tag-based encoding requires $r(3w + 8)$ characters for $w := 1$ whitespace character. A more compact sequence-style encoding would reduce this to $5r + 2$, but at the cost of ambiguity if referenced elements and values are not syntactically distinguished.

**HyMeKo**

```
@e {-a,-b,-c,...,-z}
```

**XML (nested relation elements)**

```
<e>
<r d="-">a</r>
<r d="-">b</r>
...
<r d="-">z</r>
</e>
```

**XML (attributes)**

```
<e r="a" d="-" t="b" e=">"
...
z="z" r="-"
/>
```

**YAML**

```
e:
 r: a
  d: -
 r: b
  d: -
...
 r: z
  d: -
```

**JSON**

```
{
"r": {"r":"a","d":"-"},
"r": {"r":"b","d":"-"},
...
"r": {"r":"z","d":"-"}
}
```

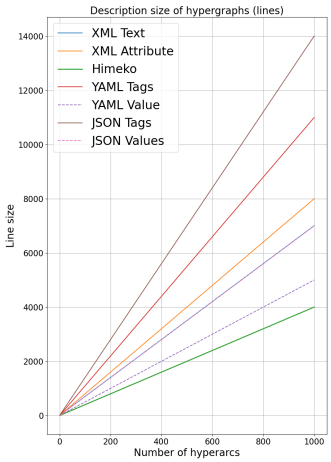
Figure 9: Representative linear hypergraph relation encodings in the compared schemas

JSON [14] is structurally closer to HyMeKo, but still requires quoted field names and quoted values. A straightforward object-based encoding requires $14r + 6$ characters, while a more compact direction-reference encoding still requires $7r + 4$ characters.

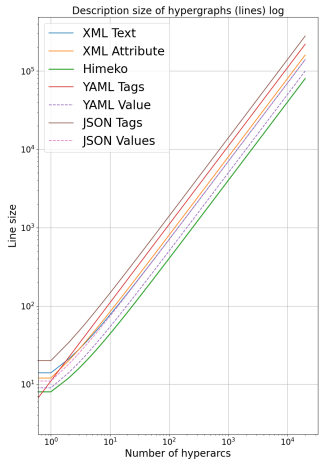
Thus, under the same simplifying assumptions, HyMeKo requires only $4r + 4$ characters and remains more compact than the corresponding XML-, YAML-, and JSON-based encodings. The resulting formulas are summarized in Table 3, and their growth is illustrated in Figure 10.

| Schema | Formula ($r$ relations) |
|---|---|
| HyMeKo (introduced schema) | $4r + 4$ |
| XML-elements | $r(2n + 10) + 7$ |
| XML-attributes | $8r + 4$ |
| YAML-tags | $11r$ |
| YAML-values | $5r + 2$ |
| JSON-tags | $14r + 6$ |
| JSON-values | $7r + 4$ |

Table 3: Character-count formulas for linear relation descriptions, assuming one-character identifiers, literals, and whitespace




(a) Increase of description length as a function of the number of hyperarcs




(b) Log plot of description length as a function of the number of hyperarcs

Figure 10: Growth of description length with respect to the number of relations

## 4.2 Structural complexity analysis

The preceding character-count analysis demonstrates syntactic compactness. However, a more meaningful comparison considers structural overhead at the graph level, namely how many auxiliary elements must be introduced solely to express a given set of relationships.

Consider a domain containing $m$ relationships, each of arity $k \geq 3$.

### 4.2.1 HyMeKo representation

In HyMeKo, each $k$-ary relationship is represented directly as a single hyperedge connecting all $k$ participants. No auxiliary elements are required:

$$\Delta V_{\text{HyMeKo}} = 0 \tag{1}$$

$$\Delta E_{\text{HyMeKo}} = m \tag{2}$$

#### 4.2.2 RDF/OWL representation

Under standard RDF reification [3], each $k$-ary relationship requires one blank node $v_r$ representing the relation instance, together with $k$ binary edges linking $v_r$ to the participating entities:

$$\Delta V_{\text{RDF}} = m \tag{3}$$

$$\Delta E_{\text{RDF}} = mk \tag{4}$$

**Proposition 1 (Structural overhead)** *Given $m$ relationships of arity $k \geq 3$, the total structural cost, measured as the sum of introduced nodes and edges, is*

$$S_{HyMeKo} = m \tag{5}$$

$$S_{RDF} = m(k+1) \tag{6}$$

*Hence, the structural overhead ratio is*

$$\frac{S_{RDF}}{S_{HyMeKo}} = k + 1 \tag{7}$$

This follows directly from the definitions of hyperedge incidence and RDF reification. For ternary relationships ($k = 3$), RDF introduces four times as many structural elements as HyMeKo; for higher arities, the gap grows linearly. This overhead affects not only storage, but also traversal and query complexity, since each auxiliary node and edge introduces additional matching steps.

Unlike the character-count comparison in Table 3, the ratio in Equation 7 depends only on relation arity and is therefore independent of surface syntax such as delimiters, quotation marks, or whitespace.

### 4.3 Illustrative benchmark scenarios

To complement the structural analysis in Equation 7, we consider two representative benchmark scenarios: a robotic kinematic model and a highly connected CPS knowledge graph. For both, we compare the structural burden of conventional encodings against HyMeKo in terms of parsed structural objects.

**Scenario A: Industrial robotics (UR5 manipulator).** The Universal Robots UR5 is a standard 6-DOF manipulator. In XML-based URDF, links, joints, visuals, and inertial properties must be declared explicitly, producing a deep and verbose abstract syntax structure. In HyMeKo, recurring link-joint patterns can be captured with structural templates («`copy`») and directed hyperarcs, reducing duplication.

**Scenario B: Humanoid robot (20-DOF symmetric).** A humanoid kinematic tree consists of 20 links and 19 revolute joints arranged in a branching structure: torso,

head (2 joints), two symmetric arms (4 joints each), and two symmetric legs (4 joints each). In URDF, every link and joint must be declared in full — including visual, collision, and inertial subelements — even when the left and right limbs are structurally identical. This produces 916 total AST nodes (380 XML elements + 536 attributes). In HyMeKo, an `arm_template` and `leg_template` are defined once and instantiated via «`copy`» for each side, overriding only the differing parameters (origin offsets, roll limits). The result is 210 AST nodes — a $4.4\times$ reduction driven primarily by template reuse.

**Scenario C: CPS sensor network.** We consider a knowledge graph containing $m = 1000$ observational events, each modeled as a 4-ary relation ($k = 4$) connecting a sensor, a spatial region, a timestamp, and a measured value. In RDF/Turtle, such events are represented through reification, requiring one auxiliary node and four binary edges per event. In HyMeKo, each observation is represented as one native 4-ary hyperedge.

**RDF/Turtle (reified)**

```
# 1 auxiliary blank node:
_:obs1 a :Observation ;
# 4 binary triples (k = 4):
  :sensor :temp_01 ;
  :region :room_A ;
  :timestamp "2026-04-01" ;
  :value 23.5 .
```

*Total:* $1 + k = 5$ *elements*

**HyMeKo**

```
# 1 hyperedge (no aux. node):
@obs1: Observation {
# 2 signed hyperarcs:
  <- temp_01,
  <- room_A,
# 2 inline fields:
  timestamp "2026-04-01",
  value 23.5
}
```

*Total:* 1 *relation element*
*(+ 2 arcs + 2 fields at AST level)*

Figure 11: A single 4-ary sensor observation in RDF/Turtle and HyMeKo. At the relation-encoding level (Proposition 1), RDF requires $k+1 = 5$ structural elements (1 auxiliary blank node + $k$ binary triples), while HyMeKo requires 1 hyperedge. The sensor and region are encoded as signed hyperarcs (<−), while timestamp and value are inline fields. For $m = 1000$ events, this compounds to 5000 vs. 1000 relation-level elements.

The resulting structural estimates are summarized in Table 4. Structural elements are counted as total parsed AST objects: XML elements plus attributes for URDF, and hypervertices, hyperedges, fields, and hyperarcs for HyMeKo.

These scenarios are consistent with the algebraic analysis: binary-edge encodings introduce linear to multiplicative structural overhead, while HyMeKo preserves a direct one-to-one representation of $n$-ary relations.

## 4.4 Comparison with semantic web and graph query standards

Building on the limitations identified in subsection 2.1, we now compare HyMeKo with widely used semantic-web and graph-query standards.

| Benchmark scenario | Format | Structural AST nodes | Overhead ratio |
|---|---|---|---|
| UR5 manipulator (kinematics) | XML (URDF) | 347 | 3.4x |
| | **HyMeKo** | **102** | **1.0x (baseline)** |
| Humanoid (20-joint) | XML (URDF) | 916 | 4.4x |
| | **HyMeKo** | **210** | **1.0x (baseline)** |
| CPS sensor network (1000 4-ary relations) | RDF/Turtle | 5000 | 5.0x |
| | **HyMeKo** | **1000** | **1.0x (baseline)** |

Table 4: Illustrative structural comparison between standard encodings and HyMeKo. For RDF/Turtle, the overhead follows $S_{\text{RDF}} = m(k+1)$.

- **RDF/Turtle**: RDF/Turtle requires reification to represent $n$-ary relations, whereas HyMeKo supports them natively.
- **OWL**: HyMeKo structural templates provide a lightweight mechanism analogous to class reuse, but without OWL-style reasoning overhead. HyMeKo does not currently target logical inference.
- **GraphQL**: GraphQL models many-to-many structures through junction types, whereas HyMeKo introduces native incidence semantics at the language level.

HyMeKo addresses the three limitations identified earlier: native $n$-ary relations, signed incidence semantics, and structural inheritance or annotation support. A summary is given in Table 5.

To illustrate the source of structural overhead, Figure 12 compares the same revolute joint in RDF/Turtle and HyMeKo. In RDF, the joint must be instantiated as a named node, with each property and participant expressed as a separate binary triple: yielding 13 structural elements. In HyMeKo, the joint is a single directed hyperedge with inline fields and signed incidence references, requiring 9 AST nodes. The difference stems entirely from reification: RDF encodes the parent–child relationship through two additional triples (`:parent`, `:child`), while HyMeKo expresses directionality natively via `<-` and `->`.

## 4.5 Example descriptions

### 4.5.1 Description of a robot

```
[ simple_robot ]
primitives {
```

| **Attribute** | **RDF/Turtle** | **OWL** | **GraphQL** | **HyMeKo** |
|---|---|---|---|---|
| Primary purpose | Knowledge representation | Ontological reasoning | Data querying | Structural modeling |
| Max. native relation arity | 2 (binary) | 2 (binary) | 2 (binary) | Unbounded |
| $n$-ary support | Reification | Class-as-relation | Junction types | Native hyperedges |
| Edge weights | Via reification | Annotation properties | Field values | Native |
| Inheritance | `rdfs:subClassOf` | Class hierarchy | Interface/Union | Structural templates |
| Incidence semantics | None | None | None | Signed $(+/-/\sim)$ |
| Parsing | $O(n)$ | Syntax-dependent | $O(n)$ | $O(n)$ LALR(1) |
| Tooling maturity | Extensive (20+ years) | Extensive (20+ years) | Extensive | Early stage |

Table 5: Comparison of HyMeKo with semantic web and graph query standards

**RDF/Turtle (reified)** — 13 structural elements

```
:shoulder_pan_joint a :RevoluteJoint
;
  :parent :base_link ;
  :child :shoulder_link ;
  :axis_x 0 ;
  :axis_y 0 ;
  :axis_z 1 ;
  :origin_x 0 ;
  :origin_y 0 ;
  :origin_z 0.089159 ;
  :limit_effort 150 ;
  :limit_lower -6.2831 ;
  :limit_upper 6.2831 ;
  :limit_velocity 3.15 .
```

**HyMeKo** — 9 AST nodes

```
@shoulder_pan: Revolute {
  axis [0,0,1],
  origin [0,0,0.089159],
  limit_effort 150,
  limit_lower -6.2831,
  limit_upper 6.2831,
  limit_velocity 3.15,
  <- base_link,
  -> shoulder_link
}
```

Figure 12: The same revolute joint expressed in RDF/Turtle and HyMeKo. RDF requires 1 node + 12 binary triples; HyMeKo uses 1 hyperedge + 6 fields + 2 directed hyperarcs.

```
    Link { name<string>, length<real> }
    @Joint { type<string>, axis<real>, range<real>, origin<real>, rpy<real> }
    @Revolute: Joint { type "revolute" }, @Fixed: Joint { type "fixed" }
}
simple_robot use <- primitives {
    base: Link { }
    link1 : Link { name "link1", length 1.0 }
    link2 : Link { name "link2", length 1.0 }
    @joint1: Fixed {<- base, -> link1}
    @joint2 : Revolute { axis [0, 0, 1], range[-3.14159, 3.14159], origin [0, 0, 1.0],
      <- link1, -> link2
    }
}
```

### 4.5.2 Description of a neural network

Listing 2: Description of a multi-layer perceptron capable learning on the Iris dataset

```
[ mlp_example import "meta_neural.himeko" ]
mlp use meta.elements {
    input_layer: I> -> linear {
        in_features 4, out_features 128
    }
    hidden_layer: I> -> linear {
        in_features 128, out_features 32
    }
    output_layer: I> -> linear {
        in_features 32, out_features 3
    }
    @connections {
        @e1: activations.relu {<- input_layer, -> hidden_layer},
        @e2: activations.relu {<- hidden_layer, -> output_layer},
        @e3: activations.softmax {<- output_layer}
    }
}
```

### 4.5.3 LLM instruction description

HyMeKo can be used to describe instructions to LLMs in a structured, yet minimal way. The following example provides a prompt, for GPT-4o by OpenAI, to collect cameras for an animal behavior monitoring setup.

Listing 3: LLM prompt

```
[chicken_camera]
cabin
{
    area 20
    ip_camera {
        ingress_protection ip65, resolution 1080p, frame_rate 30,
        protocols ["http", "rtsp"], encoding ["h264", "h265"]
    }
    camera: ip_camera {   }, camera2: ip_camera {  }, camera3: ip_camera {  }
    chicken { count 20, type "hungarian_yellow"  }
    @observes { -> camera, -> camera2, -> camera3, <- chicken  }
}
buy_camera: query {
   camera_product: match {
        ip_camera, country, price
   }
   result: tabular { link, price, product }
   limits {result{ count { min 10, max 20} } }
   }
   @search {
       <- ip_camera, <- internet, <- webshops, <- camera_product
       <- limits.result, -> result.link, -> result.price, -> result.product
       @format {  <- result }
   }
}
based in that, execute buy_camera with limits, and format
```

The LLM was capable to understand the request to search for outdoor cameras (with ingress protection IP65), at least 1080p resolution. In the end, the LLM provided meaningful, structured output for the query (as depicted in Listing 4).

Listing 4: LLM result

```
Searched 4 sites
I found several IP cameras with the specifications you need (IP65, 1080p resolution, 30fps, H.264/H.265
    ↪ encoding)
    available on online marketplaces.
```

```
Searched 3 sites
Product Price Link
Amcrest ProHD ADC2W $74.99 Amcrest ADC2W
Amcrest SmartHome ASH26-W $169.99 Amcrest ASH26-W
Wyze Cam Pan v3 $39.98 Home Depot
Wyze Outdoor Cam $59.99 Best Buy
EZVIZ LC1C Full HD $91.41 Amazon Search
ZOSI 1080P H.265+ System Varies Amazon Search
```

**Conclusions** This article presented HyMeKo, a formal markup language for highly connected data based on hypergraph theory. The language organizes information in a hypertree-based structure where hyperedges natively express n-ary, directed relationships with signed incidence semantics, eliminating the reification overhead required by binary-edge formats such as RDF and OWL. Template-based modeling and structural inheritance enable modular, reusable descriptions without external tooling. A structural complexity analysis (Proposition 1) shows that for $k$-ary relationships, binary-edge representations introduce a $(k+1)$-fold overhead in auxiliary nodes and edges compared to HyMeKo. Three illustrative benchmarks confirmed this in practice: a UR5 manipulator exhibited $3.4\times$ overhead in URDF, a 20-DOF humanoid with symmetric limbs showed $4.4\times$ due to template reuse, and a CPS sensor network required $5.0\times$ more structural elements in RDF/Turtle. Reference implementations exist in Python (PyLark) and Rust (LALRPOP), the latter accelerated with a SIMD-based lexer and PyO3 bindings, demonstrating the language's portability across parser frameworks. Future work will focus on a native query engine for hypergraph traversal and pattern matching, as well as interoperability layers for RDF import and export.

**Acknowledgements** Supported by the EKÖP-24-4-I-SZE University research fellowship program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

## References

[1] F. DeRemer and T. Pennello, "Efficient Computation of LALR(1) Look-Ahead Sets," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 615–649, 1982. Place: New York, NY, USA.

[2] H. S. Thompson and C. Lilley, "XML Media Types," 2014. Issue: 7303 Num Pages: 35 Series: Request for Comments Published: RFC 7303.

[3] N. Noy and A. Rector, "Defining N-ary Relations on the Semantic Web," working Group Note, W3C, 2006.

[4] F. Orlandi, D. Graux, and D. O'Sullivan, "Benchmarking RDF Metadata Representations: Reification, Singleton Property and RDF," in *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, pp. 233–240, 2021.

[5] V. Nguyen, O. Bodenreider, and A. Sheth, "Don't like RDF reification? making statements about statements using singleton property," in *Proceedings of the*

*23rd International Conference on World Wide Web*, WWW '14, (New York, NY, USA), pp. 759–770, Association for Computing Machinery, 2014.

[6] A. G. Salguero, C. Delgado, and F. Araque, "Easing the Definition of N–Ary Relations for Supporting Spatio–Temporal Models in OWL," in *Computer Aided Systems Theory - EUROCAST 2009* (R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, eds.), (Berlin, Heidelberg), pp. 271–278, Springer Berlin Heidelberg, 2009.

[7] N. Koenig and A. Howard, "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator," in *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2149–2154, 2004.

[8] D. Tola and P. Corke, "Understanding URDF: A Survey Based on User Experience," 2023. _eprint: 2302.13442.

[9] N. Albergo, V. Rathi, and J.-P. Ore, "Understanding Xacro Misunderstandings," *ArXiv*, vol. abs/2109.09694, 2021.

[10] O. Michel, "Webots: Professional Mobile Robot Simulation," *Journal of Advanced Robotics Systems*, vol. 1, no. 1, pp. 39–42, 2004.

[11] A. Bretto, *Hypergraph Theory: An Introduction*. Springer Publishing Company, Incorporated, 2013.

[12] Q. Dai and Y. Gao, "Mathematical Foundations of Hypergraph," in *Hypergraph Computation*, pp. 19–40, Singapore: Springer Nature Singapore, 2023.

[13] R. Polli, E. Wilde, and E. Aro, "YAML Media Type," 2024. Issue: 9512 Num Pages: 13 Series: Request for Comments Published: RFC 9512.

[14] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," 2017. Issue: 8259 Num Pages: 16 Series: Request for Comments Published: RFC 8259.