Processing TIL-Script Constructions for Supervised Machine Learning with Symbolic Representation

Tomáš Michalovský[1], Marek Menšík[1] and Adam Albert[2]

[1]Silesian University in Opava, Bezručovo nám. 13, 746 01 Opava, Czech republic
[2]VSB - Technical University of Ostrava, 17. listopadu 15, Ostrava, Czech republic

Abstract
This paper deals with the processing of TIL-Script constructions. TIL-Script constructions are a computational variant of the Transparent Intensional Logic (TIL) constructions. In this paper, two main directions are addressed. The first area is the use of TIL-Script in supervised machine learning with a symbolic representation of facts in a search for explications of atomic concepts. Subsequently, the paper discusses the modifications that were necessary for use in the spatial data processing. The paper describes the theory necessary to understand TIL theory, Supervised machine learning, formal conceptual analysis, and spatial data processing issues specified in the TIL-Script language.

Keywords
Machine learning, Refinement, Generalization, Specialization, Hypothesis, Heuristics

1. Introduction

This paper outlines the use of Transparent Intensional Logic (TIL) constructions with a focus on Formal Conceptual Analysis (FCA) and supervised machine learning. The paper is divided into two parts. The first part contains the use of machine learning and FCA to find relevant textual sources of information. The second part contains the modification of supervised machine learning to process constructions that represent spatial data.

The following section summarizes the current developments in the use of TIL and, consequently, TIL-Script in supervised machine learning.

The paper is structured as follows. In the Chapter 2, the reader is introduced to the basics of TIL. This chapter is taken as a summary of the most important characteristics and definitions that are relevant for understanding the paper. However, since the language of TIL constructions is quite complex to process by a computer, in the Chapter 2.1, we introduce the *TIL-Script* language, which is a computational variant of TIL, and which is used in machine processing. In the Chapter 4, we will discuss the basics of formal conceptual analysis theory and the use of this method to find relevant information sources. Subsequently, the Chapter 5 discusses the issue of working with spatial data.

ITAT'22: Information technologies – Applications and Theory, September 23–27, 2022, Zuberec, Slovakia
michalovsky810@gmail.com (T. Michalovský);
mensikm@gmail.com (M. Menšík); adam.albert@vsb.cz (A. Albert)

2. TIL

This section briefly introduces the Transparent Intensional Logic system.

TIL is a partial, typed hyperintensional λ-calculus of partial functions with procedural semantics. Expressions of natural language encode *algorithmically structured procedures* as their meaning. These procedures produce extensional or intensional entities, or even lower-order procedures, as their products. In the early 1970s, Pavel Tichý defined six kinds of such procedures, which he coined TIL *constructions* as the centerpiece of his system; see [1].

The TIL distinguishes between two types of construction: atomic and molecular constructions. These constructions are *Trivialization* and *Variables*. The operational meaning of *Trivialization* is similar to the meaning of constants in formal languages or pointers in programming language terminology. Trivialization supplies an object O without mediating any additional procedures. We write the trivialization of an object O as $'O$. The second atomic construction, *Variable*, produces objects depending on its evaluation. The variable v-constructs an object.

The following constructions are molecular. *Composition* $[F\ A_1 \dots A_n]$ corresponds to an application in λ-calculus. It is a procedure for applying a function f to an ordered tuple (if any) produced by the $A_1 \dots A_n$ procedure. The v-composition constructs the value of the function f on the arguments of $A_1 \dots A_n$ if the function f is defined on the arguments of $A_1 \dots A_n$. However, if the value of the function f on the arguments $A_1 \dots A_n$ is not defined, then *Composition* is v-improper (fails), i.e. v-constructs nothing. This situation can also occur if there is a type of inconsistency in the construction. The

Closure $[\lambda x_1 \ldots x_n\ X]$ corresponds to a λ-abstraction in a λ-calculus. It is a procedure that v-constructs a function by abstraction over the values of the variables $x_1, \ldots, x_n$. The closure is never v-improper for any valuation, since it always v-constructs a function.[1] Now we define *Double execution*, *Single execution* is not relevant to us now. The *Double Execution X* will execute the given construction twice, thus reducing, for example, the mode of occurrence of the *display* construction to the mode of the *execution* construction. If X v-constructs a Y construction, and Y v-constructs an entity Z, then X v-constructs Z. Otherwise, X is v-improper in that it does not produce anything.

The TIL ontology is organized into a branching hierarchy of types built on top of the base. It is a two-dimensional type hierarchy. The first "horizontal" dimension increases the degree of molecularity, i.e. it starts at the level of atomic objects (base objects) and builds a hierarchy of functions above them by folding functions into more and more complex functions. The second, "vertical" dimension increases the order of construction. At its beginning are entities that are not constructions, followed by constructions that construct non-constructions, etc. At the base level of type hierarchies, there are, from an algorithmic point of view, nonconstructive entities belonging to *types of order 1*. Given the base of *atomic types* (o-truth values, ι-individuals, τ-time instants/real numbers, ω-possible worlds), the induction rule for function creation is applied: where $\alpha, \beta_1, ..., \beta_n$ are *types of order 1*, the set of partial mappings from $\beta_1 \times ... \times \beta_n$ to α, denoted by $(\alpha\ \beta_1 ... \beta_n)$ is also an *order 1* type. Constructions that construct entities of order 1 type are *order 1* constructions. They belong to the order 2 type denoted by $*_1$. This type, together with atomic types of order 1, serves as the basis for the induction rule: any set of partial mappings, type $(\alpha\ \beta_1 ... \beta_n)$, involving $*_1$ in their domain or range is a *type of order 2*. Constructions that construction entities of type order 1 or 2 are *constructions of order 2*. They belong to *type of order 3* denoted by $*_2$. Any set of partial mappings involving $*_2$ in their domain or range is a *of type order 3* and so on ad infinitum.

2.1. TIL-Script

TIL is not suitable for computer processing due to its complex notation; therefore, we are using its computational variant, TIL-Script [13]. TIL-Script offers standardized notation, base of types, and form notation using ASCII characters only (letters of the Greek alphabet, indices, etc. cannot be written using ASCII characters). Compared to TIL, TIL-Script has a broader base of types, making construction more easily handled compared to TIL. Some of the newly added types are well known for common programming languages (Bool, Int, Real) and others are native to TIL-Script (Indiv, World, Time). Each TIL construction has an equivalent in TIL-Script, but with a different syntax. For comparison, here are some examples:

- Trivialization in TIL: ${}^0C \rightarrow$ Trivialization in TIL-Script: 'C
- Composition in TIL: $[F\ A_1 \ldots A_n] \rightarrow$ Composition in TIL-Script: $[F\ A1 \ldots An]$
- Closure in TIL: $[\lambda x_1 \ldots x_n\ X] \rightarrow$ Closure in TIL-Script: $[\backslash x1 \ldots [\backslash xn\ X] \ldots]$

An example of a natural language sentence captured in TIL and TIL-Script:

- Sentence: Charles counts 3 + 5.
- TIL: $\lambda w \lambda t[{}'Counts_{wt}\ {}'Charles\ [{}'+\ {}'3\ {}'5]]$
- TIL-Script: $[\backslash w\ [\backslash t\ [{}'Counts@wt\ {}'Charles\ [{}'+\ {}'3\ {}'5]]]]$.

There is an EBNF grammar for *TIL-Script*.

3. Supervised machine learning

In [2], we describe machine learning with the teacher. Here, we briefly summarize the basic characteristics of this approach.

Supervised machine learning is a subcategory of machine learning that uses training data partitioned into positive and negative examples of a concept to be learned. These data are described by a set of input and output attributes. There is an unknown functional dependency f between the values of the input and output attributes. The goal of learning is to approximate the unknown functional dependency f with the functional dependency h called *hyphotesis*. The hypothesis is obtained by observing the values of the input and output attributes of the training data. The correctness of the hypothesis is tested on test data for which only the input attribute values are known to the learner. The hypothesis is correct if it correctly predicts the values of the output attributes of the test data.

Supervised machine learning is most commonly used to solve two types of problem: *regression* and *classification*. *Regression* is a problem in which the values of the output attributes are elements from a continuous range of numbers. For example, when predicting the price of a diamond based on its properties. Examples of regression algorithms are *linear regression*, *logistic regression*, or *polynomial regression*. In *classification*, the output values are discrete, and the model matches the input examples with the output categories. The classification problem is, for example, the recognition of traffic signs in the image data. Common classification algorithms are *Support Vector Machine*, *decision trees*, and *random forest*.

[1] This is a so-called *generated function that v-constructs nothing for each tuple*

3.1. Winston

Patrick Winston's algorithm [14] presents learning using positive and negative *near-miss* examples. A near-miss example represents a negative example that differs from a positive example in one important difference. It is a supervised algorithm; therefore, each input example is labeled as positive or negative, and the model (hypothesis in the learning process) is modified based on observations of the values of the input and output attributes of these examples. The model is modified using induction heuristic functions. The heuristic functions are **require-link**, **forbid-link**, **climb-tree**, **enlarge-set**, **drop-link** and **close-interval**. Winston represented the examples graphically by a semantic network in which the attribute values of objects are represented as nodes of the network, and the relations between them (edges of the network) are called *links*.[2] The heuristics are described as follows.

- **require-link**: Used if the model contains *link*, which the near-miss example does not. The given *link* is marked in the model as *MUST-BE* (must be).
- **forbid-link**: Used when there is a *link* in the near-miss example that is not in the model. The given *link* is inserted and marked as *MUST-NOT-BE* (must not be) in the model.
- **climb-tree**: Used when we want to generalize a too specific model. Based on the positive example, we generalize an attribute value that differs in the model. The most common general value of the attribute of the model and the example found in the ontology provided by the teacher replace the value in the model.
- **enlarge-set**: Used if the ontology is not provided or the values in the example and model do not share a common most specific value in the ontology. In this case, the values are concatenated into a set of values.
- **drop-link**: Used when the model has a *link* where the positive example does not, or the attribute values are mutually exclusive. The given link is removed from the model.
- **close-interval**: Used when a numeric value or an interval of numeric values is present in the positive example. If the values from the positive example are not already included in the model, the interval in the model is extended to include these new values.

Winston defines two main methods, *Generalization* and *Specialization*, in which heuristics are applied to the model.

Specialization is triggered using near-miss examples.

1. Compare the hypothesis model and the near-miss example to find a significant difference.
2. If a significant difference exists, then proceed:
   a) If the model has a link and the near-miss example does not, then **require-link** is used.
   b) If the near-miss example has a link and the model does not, **forbid-link** is used.
   c) Otherwise, the example is ignored.

Generalization is done with positive examples.

1. Compare the hypothesis model and the positive example to find the difference.
2. For each difference is verified:
   a) If the link in the model binds to a value different from the value in the example, then it is verified:
      i. If the attributes in which the model and the example differ have the most specific generic value, then **climb-tree** is used.
      ii. If the attribute values are inconsistent, **drop-link** is used.
      iii. Otherwise, **enlarge-set** is used.
   b) If there is a link in the model that is not in the example, **drop-link** is used.
   c) If the model and the example differ in a numeric value or an interval of numeric values, **close-interval** is used.
   d) If none of the above can be applied, the example is ignored.

3.2. Natural Language Processing

This work is based on the use of TIL in natural language processing, described in detail in [4]. For completeness, we describe the whole concept. The algorithms described in the 5 section build on these ideas.

In [4], a method that supports automatic information retrieval was proposed to select a relevant information source from many potentially relevant ones. The method is based on the logical analysis of natural language texts in the form of a TIL constructions language. Combined with a machine learning algorithm based on the aforementioned Winston's algorithm, explanations of atomic concepts are extracted from the formalized texts.[3][4] Based on these explications and user preferences, the most relevant sources of information from which the explications were built are recommended.

[2] For example, if a cube is white in color, there would be two nodes in the graphical representation, one for the object *cube* and one for the attribute value *white*. These nodes would be connected by *link* has_{color}

[3] A concept in TIL terminology is a closed construction.

[4] Carnapian explication, used in this paper, is the process of refining a vague or inaccurate expression into an adequately accurate one. For simplicity, we refer to the refinement of a given expression as an explication.

3.2.1. Modified Winston's algorithm

In order to extract concept explications from text sources, Winston's algorithm had to be adjusted. The input of the algorithm (examples) is formalized natural language sentences that mention the concept to be explicated. These sentences are formalized into TIL constructions. The result of the algorithm is a molecular TIL construction that describes a simple concept. In addition to the *generalization* and *specialization* methods, the algorithm also includes a *refinement* method that inserts new constituents into the molecular construction.

Natural language sentences often contain only partial information about a given simple concept. The examples used by Winston always contain complete information. For this reason, a new heuristic method *Concept-introduction* was proposed, which adds partial information as constituents to the explication in a refinement process. On the basis of the negative examples, the *Negative-concept* method is triggered, which adds new constituents to the explication in a negated way. This method is used to distinguish explications of similar concepts. Generalization replaces the values of the constituents with more general values.

Generalization contains heuristic functions *General-concept*, *Disjunctive-concept* and *Close-interval*. First two heuristic functions has similar functionality. They generalize values in constituents of the model. If the supervisor provides an ontology of values, and the positive example differ in value from the model, *General-concept* replaces the value in the model with the most specific general value of values in the model and example from ontology. If the ontology is not provided, *Disjunctive-concept* generalize the value in the model with an union of values in the model and example.

If the model and example differ in numerical value *Close-interval* generalize the model's value with the numerical interval spanning both values.

The algorithm is described in detail in [4].

4. Formal Conceptual Analysis

One of the main focuses of this work is the application of *Formal conceptual analysis* (FCA) to the explications obtained, *aspirants ordering* and the implementation of both methods.

In this section, we discuss a procedure for finding a suitable atomic concept using explications of simple concepts and FCA.[5] This procedure has been published in [8]. The output of the method is a recommendation of a suitable concept based on a given set of properties or attribute values that occur in the explications obtained. Using FCA, we search for a concept that is represented by a given set of properties or attribute values.

FCA [15] was introduced by Rudolf Wille in 1981. It is a popular technique used for data mining, software engineering, knowledge processing, machine learning, and many others. FCA studies the relationships between objects described by a set of attributes and their hierarchical grouping based on common attributes.

We now describe FCA and *aspirant ordering* using formal definitions presented in [5] and then explain the methods by example.

Definition 1. *Let (G, M, I) be a formal context, then $\beta(G, M, I) = \{(O, A) | O \subseteq G, A \subseteq M, A^{\downarrow} = O, O^{\uparrow} = A\}$ is a set of all formal concepts of context (G, M, I) where $I \subseteq G \times M, O^{\uparrow} = \{a | \forall o \in O, (o, a) \in I\}, A^{\downarrow} = \{o | \forall a \in A, (o, a) \in I\}$. $A^{\downarrow}$ is called **extent** of the formal concept (O, A) and $O^{\uparrow}$ is called **intent** of the formal concept (O, A).*

Definition 2. ***Concept aspirants*** *of the set of attributes **a** in $\beta(G, M, I)$ is a set $CA(a) = \bigcup_{i=1}^{n} O_i^a$, where O^a is the extent of a concept $(O, A) \neq (G, B), a \subseteq A, B \subseteq M$. Namely, **concept aspirants** of the set of attributes **a** is a union of all formal concept extents where **a** is a subset of a particular formal concepts' intents.*

Definition 3. *Let $CA(a)$ be a set of concept aspirants of a set of attributes **a**, let $\delta(a)$ be a set of concepts (O, A) where $a \subseteq A$, i.e.: $\delta(a) = \{(O^a, (O^a)^{\uparrow}) | (O^a, (O^a)^{\uparrow}) \neq (G, B), B \subseteq M, (O^a, (O^a)^{\uparrow}) \in \beta(G, M, I)\}$. Then $\boldsymbol{x} \sqsubseteq \boldsymbol{y}$ is in relation of **aspirant ordering** iff $max(|(O^y)^{\uparrow}|) \leq max(|(O^x)^{\uparrow}|), x, y, \in CA(a), (O^x, (O^x)^{\uparrow}), (O^y, (O^y)^{\uparrow}) \in \delta(a)$.*

Definition 4. *Let $(CA(a), \sqsubseteq)$ be an ordered set according to the definition 3, then the maximal elements are **most appropriate concepts**.*

FCA is used to obtain the set of all formal concepts and form a conceptual lattice over explications. The conceptual lattice captures the hierarchical ordering of the explications. On the basis of the set of all formal concepts, we find the so-called *Concept Aspirants*. *Concept Aspirants* is the union set of all *intents* of formal concepts in which a set of selected attributes occurs. We order this set according to the definition of 3. If the set is in the relation *Aspirant Ordering*, then its maximal element is the **most appropriate** concept describing the search object.

4.1. Program implementation

The entire algorithm for FCA and *Aspirant Ordering* is implemented in Java without using third-party libraries that address the issue. The formal context, which is the FCA input, is displayed in the code 1 and is obtained from the input CSV file. Each row of the formal context

[5] For example, if we know some basic characteristics of an object for which we do not know the name.

represents one object, and the values 1 or 0 in the columns indicate whether the object has the attribute. The next input argument is the attributes based on which the most appropriate concept will be searched.

```
[O/A,  a1,  a2,  a3,  a4,  a5,  a6,  a7,  a8,
    ↪ a9,  a10]
[o1,    1,   1,   1,   1,   1,   1,   1,   1,
    ↪ 0,   0]
[o2,    0,   0,   0,   0,   0,   0,   0,   0,
    ↪ 1,   1]
[o3,    0,   0,   0,   0,   0,   0,   0,   0,
    ↪ 0,   0]
[o4,    1,   1,   0,   0,   0,   0,   1,   0,
    ↪ 0,   0]
[o5,    0,   0,   0,   0,   1,   0,   0,   0,
    ↪ 0,   0]
[o6,    0,   0,   0,   0,   1,   0,   0,   0,
    ↪ 0,   0]
[o7,    1,   1,   1,   0,   0,   0,   0,   0,
    ↪ 1,   0]
```

Code 1: Input data

First, we need to perform a complete FCA according to the definition of 1. We are looking for sets of objects that share sets of attributes. Each formal concept corresponds to a maximal submatrix whose elements all have the same attributes equal to 1. However, objects that form a formal concept do not need to have identical set of attributes, e.g., an object o_2 and an object o_7 can form a formal concept $C_1 = (\{o_2, o_7\}, \{a_7\})$ even if o_7 has attributes that o_2 does not. The FCA algorithm is defined as follows:

1. For every possible set of objects:
    a) For each attribute:
        i. Check if each object of the currently examined set has the given attribute:
            A. If so, the attribute is added to the set of common attributes.
            B. If not, the attribute is ignored.
    b) Check if a formal concept with an identical set of common attributes already exists:
        i. If so, compare the number of objects of the formal concept and the currently examined set of objects:
            A. If the currently examined set has more objects than the existing formal concept, the existing formal concept is replaced by a new one including more objects.
            B. If the existing formal concept has more objects, no change is made.
        ii. If not, a new formal concept is created with the currently examined set of objects and set of common attributes.

```
...
//if set of attributes was not added yet, new
    ↪ formal concept with this set of
    ↪ attributes will be added
if(!subsetsOfAtt.contains(commonAtt)){
    subsetsOfAtt.add(commonAtt);
    FormalConcepts.add(new FormalConcept(
    ↪ setOfSubsets.get(subsetNo), commonAtt));
}
    //if set of attributes is already included
    ↪ in one of the formal concepts, then sizes
    ↪  of sets of objects are compared and the
    ↪ one with more elements is added into set
    ↪ of formal concepts
else  if(FormalConcepts.get(subsetsOfAtt.indexOf
    ↪ (commonAtt)).getObjects().size() <
    ↪ setOfSubsets.get(subsetNo).size()){
    int index = subsetsOfAtt.indexOf(commonAtt);
    FormalConcepts.remove(index);
    subsetsOfAtt.remove(index);
    subsetsOfAtt.add(commonAtt);
    FormalConcepts.add(new FormalConcept(
    ↪ setOfSubsets.get(subsetNo), commonAtt));
}
...
```

Code 2: Adding a new formal concept

If we have a complete FCA, then the algorithm for *Aspirant Ordering* can start. In this particular example, the input will be a_1 and a_2, for which we want to find the most suitable concept that is described by the attributes given. We first look for formal concepts in which both input attributes occur at the same time; formal concepts are listed in code 3.

```
...
for(FormalConcept CurrentFormalConcept :
    ↪ FormalConcepts) {
    if(CurrentFormalConcept.containsAttribute(
    ↪ attributes))
        ConceptsWithSoughtAtt.add(
    ↪ CurrentFormalConcept);
}
...

Formal concepts with attributes [a1, a2]:
[] = [a1, a2, a3, a4, a5, a6, a7, a8, a9, a10]
[o1] = [a1, a2, a3, a4, a5, a6, a7, a8]
[o1, o4] = [a1, a2, a7]
[o7] = [a1, a2, a3, a9]
[o1, o7] = [a1, a2, a3]
[o1, o4, o7] = [a1, a2]
```

Code 3: Formal concepts featuring input attributes

In the next step, we get the union of all extents from the previous step; as shown in code 4. We get all objects that have both attributes that we are looking for.

```
...
for(FormalConcept CurrentFormalConcept :
    ↪ ConceptsWithSoughtAtt) {
    for(String extent : CurrentFormalConcept.
    ↪ getObjects()){
        Extents.add(extent);
    }
}
...

Extents for [a1, a2]:
[o1, o4, o7]
```

Code 4: Union of extents

Next, for each of the retrieved objects, we search for its intent, as shown in code 5. This gives us a set of objects with all their attributes, where one of them is the most appropriate concept to be searched for. This set is called *Concept Aspirants* by the definition 2.

```
...
for(String object : Extents){
    HashSet<String> Attributes = new HashSet<>()
    ↪ ;
    for(FormalConcept CurrentFormalConcept :
    ↪ FormalConcepts) {
        if(CurrentFormalConcept.containsObjects(
    ↪ object)){
            for(String attributeToAdd :
              CurrentFormalConcept.getAttributes
    ↪ ())
                Attributes.add(attributeToAdd);
        }
    }
    List<String> AttributesList = new ArrayList
    ↪ <>(Attributes);
    List<String> ObjectList = new ArrayList<>();
    ObjectList.add(object);
    ConceptAspirants.add(new FormalConcept(
    ↪ ObjectList, AttributesList));
}
...

Concept aspirants
[o1] = [a1, a2, a3, a4, a5, a6, a7, a8]
[o4] = [a1, a2, a7]
[o7] = [a1, a2, a3, a9]
```

Code 5: Set of Concept Aspirants

In the last step, we sort the set of *concept aspirants* according to the definition 3 and select the maximum element to obtain the most suitable concept described by the input attributes.

```
Aspirant ordering
[o4] = [a1, a2, a7]
[o7] = [a1, a2, a3, a9]
[o1] = [a1, a2, a3, a4, a5, a6, a7, a8]

The most appropriate concept described by
    ↪ attributes [a1, a2] is:
[o4] = [a1, a2, a7]
```

Code 6: Ordering of concepts and the most appropriate concept

5. Heuristics for spatial data

In this section of the paper, the issue of knowledge base representing space in multi-agent systems is discussed. Using natural language processing, TIL, and previously described machine learning methods, we create a topological representation of a map that an agent can use to navigate in space. The first outline of this approach is proposed in [9].

The input for the construction of the topological map representation is a set of ordered sentences formalized using TIL-Script constructions describing the agent's journey through the environment (in this particular case, a path through a city). Each sentence may contain information about who went where, how, from where, through what, etc., but some information may sometimes be missing or incomplete. We assume that the different parts of the journey are described sequentially, and hence some missing information can be obtained from the previous sentences. However, once a representation of one path is constructed, its form may not be final but may change if, for example, the same path is described in more detail by another agent; hence, we are concerned not only with the construction of the path representation but also with its modification. Part of this paper deals with the problem of knowledge base representing space in multi-agent systems.

The input to the algorithm is an ordered set of natural language sentences formalized into TIL-Script. These sentences describe the journey of some agent through the environment (in this particular case, a journey through a city). The sentences contain information about who, how, from where, to where, through what, etc. traveled. However, not all sentences contain complete information. The missing information is filled in from previous sentences in the preprocessing of the input, as it is a sequential description of the journey. The machine learning algorithm then creates a molecular construction that describes the agent's journey from the preprocessed input. By combining the different paths of the agents with the same algorithm, we then obtain a topological representation of the space in which the agents traveled. In this paper, however, we focus only on the creation of a single agent path.

The heuristic methods and algorithm described in Section 3.2, but modified to handle path descriptions, are the basis for creating space representations. The fundamental difference from the application of the algorithm already mentioned is that we are now not specifying a simple concept, but building a molecular construction describing the agent's path.

To handle path descriptions, we use the class of so-called motion verbs (which contains e.g. the verbs go, run, ride, cross, turn, etc.). Verbs in this class bind certain parts of sentences to each other by valency. The

valency bindings for individual verbs are described using valency frames. The valency frame provides information on which complements are valence-bound to a particular motion verb [16]. For example, the verb *go* is valence-bound in sentences by complements written with a functor, such as *ACT* (who went), *DIR1* (where he went), *DIR2* (which way/what he went), *DIR3* (where he went), *MANN* (how).

In [9] we used the following:

- Actor [ACT] - **Paul** quickly walked home from school through the city center.
- Direction - from where [DIR1] - Paul walked quickly home **from school** through the city centre.
- Direction (direction) - which way [DIR2] - Paul walked quickly home from school **through the city centre**.
- Direction (direction) - where [DIR3] - Paul walked quickly **home** from school through the city centre.
- Manner (way) [MANN] - Paul walked **fast** home from school through the city centre.
- Extent [EXT] - After **100m** he turned right.

Using the valency structures of motion verbs, we get a description of the journey. The path locations, identified by directional functors (DIR1, DIR2 and DIR3), are the basic information for the construction of the path description. The path representation is defined in [7] using the following terms.

Definition 5 (*node, edge*). *Let V be a motion verb, let $S = \{B|('DIR1 \text{ or } 'DIR3) \text{ and } V \text{ are constituents of } B\}$ and let $D^V = \{C|V \text{ is a constituent of } C\} \setminus S$ then D^V is a set of **edges** and S is a set of **nodes**.*

Definition 6 (*place, functor, value*). *Let $[\alpha\ x\ v]$ be a node and let $[\beta\ y\ v_1]$ be an edge. Then x is a **place**, α, β are **functors**, and y, v, v_1 are **values**.*

Definition 7 (*connection*). *Inductive definition:*

1. *Let α be an edge then α is a connection (atomic).*
2. *Let α, γ be connections, let β be node then $\alpha \rightarrow \beta \rightarrow \gamma$ is a **connection**.*
3. *Only structures in 1 and 2 are connections.*

Remark: Connection is a transition between nodes.

Definition 8 (*path*). *Let α and γ be nodes and let β be a connection then $\alpha \rightarrow \beta \rightarrow \gamma$ be a **path**.*

Definition 9 (*same path*). *Let $\alpha \rightarrow \beta \rightarrow \gamma$ and $\alpha \rightarrow \delta \rightarrow \gamma$ be paths in model and positive example respectively, let $\delta = \epsilon \rightarrow ... \rightarrow \omega$ then if $\lambda x['DIR2_{wt}\ x\ y]$ in β (model) = $\lambda y['DIR2_{wt}\ y\ z]$ in δ (positive example) then both paths represent the **same path**.*

Definition 10 (*network*). *A network is a set of all paths.*

5.1. Path Preprocessing

As mentioned earlier, input sentences may not always contain all the necessary information. If node information is missing, then it is not *path* as defined by definition 8. However, the heuristics described in the following sections can only be applied to *paths*, so we must first deploy the *Path Preprocessing* algorithm.

```
...
if(!isPath){
    if(isSen1){
        if(hasDir1){
            if(hasDir3)
                pass = true;
            else{
                processed_sentence.add(addDIR3(
    ↪ motion_verb, dummy_val));
                pass = true;
            }
        }
        else{
            pass = false;
        }
    }
    else{
        if(hasDir1){
            if(hasDir3)
                pass = true;
            else{
                processed_sentence.add(addDIR3(
    ↪ motion_verb, dummy_val));
                pass = true;
            }
        }
        else{
                processed_sentence.add(addDIR1(
    ↪ motion_verb, previous_sentence));
            if(!hasDir3){
                processed_sentence.add(addDIR3(
    ↪ motion_verb, dummy_val));
                pass = true;
            }
            else
                pass = true;
        }
    }
}
 else
    pass = true;
...
```

Code 7: Part of Path Preprocessing algorithm

5.2. Heuristic application

The *Path Introduction* and *Path Update* heuristics described in the following sections are applied in different cases. Before deploying the heuristics, we need to check whether the model and the example capture *same paths* as defined by definition 9, and select the appropriate heuristic to apply based on the result. If they are *Same Paths*, the *Path Update* heuristic is applied. Otherwise, *Path Introduction* is applied. The description of the algorithm for checking *Same Paths* is as follows:

1. Check if DIR1 locations are the same in the model and example:
    a) If not, the algorithm terminates, and they are not *same paths*.
    b) If so, the algorithm continues.
2. Check if the DIR3 locations in the model and example are the same:
    a) If not, the algorithm ends and they are not *same paths*.
    b) If yes, the algorithm continues.
3. Check if all nodes in the model and example have the same DIR2 locations:
    a) If not, the algorithm ends and they are not *same routes*.
    b) If yes, the algorithm continues.
4. All conditions are met and they are a *same paths*.

Subsequently, a specific heuristic is already applied.

5.3. Path Introduction

After passing all sentences through *Path Processing* and checking whether they are *same paths* or not, we can start building the actual topological representation of the map from these sentences. First, we focus on the *Path Introduction* heuristic. This heuristic works with two constructions: the existing path model and an example representing a sub-path. *Path Introduction* adds a new *path* to the model. We will explain the whole process using an example.

```
j1 = [[['ACT, 'Peter, 'went], ['DIR1, 'A, 'went
    ↪ ], ['DIR2, 'X, 'went],
['DIR3, 'B, 'went]]]

j2 = [[['ACT, 'Peter, 'went], ['DIR1, 'B, 'went
    ↪ ], ['DIR2, 'X, 'went],
['DIR3, 'C, 'went]]]
```

Code 8: Path Introduction input data

The heuristic algorithm proceeds as follows:

1. From the positive example, we get all explications representing nodes (containing DIR1 or DIR3).
2. For each explication of the model:
    a) If DIR1 or DIR3 appears in the current explication, then:
        i. Loop through all the nodes obtained in step 1 from the example.
        ii. Compare the location in the example node with the location in the currently examined model node:
            A. If they match, we mark the example as connected to the the model.
            B. If they do not match, continue by examining other nodes.
3. If the example was marked as connected, then the entire example is added to the model.

Branch (a) of the above algorithm description is displayed in the code 9.

```
...
for(List<String> example_location :
    ↪ example_nodes){
    if(example_location.get(1).equals(model.
    ↪ getPathExplication().get(x).get(y).get(1)
    ↪ )){
            connected = true;
            break;
    }
}
...
```

Code 9: Part of Path Introduction algorithm

5.4. Path Update

The second heuristic for building the topological representation of the map is *Path Update*, which modifies the existing *path* in the model. The heuristic also works with a model and a positive example and is applied when both *paths* represent the *same path*. The positive example then describes the given *path* in more detail than is captured in the model, so we use the information from the positive example to refine the model.

Let us have the input data displayed in code 10. Both *paths* have the same origin and destination and also identical DIR2 locations, so *Path Update* can be deployed.

```
model = [[['ACT, 'Dan, 'walking], ['DIR1, 'A, '
    ↪ walking],
    ['DIR2, 'X, 'walking], ['DIR3, 'D, 'walking
    ↪ ]]]

positive_example = [[['ACT, 'Peter, 'went], ['
    ↪ DIR1, 'A, 'went],
    ['DIR2, 'X, 'went], ['DIR3, 'B, 'went]], [['
    ↪ ACT, 'Peter, 'went],
    ['DIR1, 'B, 'went], ['DIR2, 'X, 'went], ['
    ↪ DIR3, 'C, 'went]],
    [['ACT, 'Peter, 'went], ['DIR1, 'C, 'went],
    ↪ ['DIR2, 'X, 'went],
    ['DIR3, 'D, 'went]]]
```

Code 10: The model and positive example for Path Update

We refine the model by removing the entire DIR3 node explication, since here we have to link to the new location from the positive example. We then merge the model and positive example explications and insert the previously removed DIR3 node explication at the end. The resulting model is displayed in the code 11.

```
model = [[[['ACT 'Dan 'walking] ∨ ['ACT 'Peter '
    ↪ went]] ∧
    [['DIR1 'A 'walking] ∨ ['DIR1 'A 'went]] ∧
    ↪ [['DIR2 'X 'went] ∨
    ['DIR2 'X 'walking]] ∧ ['DIR3 'B 'went]] ∧
    ↪ [['ACT 'Peter 'went] ∧
    ['DIR1 'B 'went] ∧ ['DIR2 'X 'went] ∧ ['DIR3
    ↪  'C 'went]] ∧
    [['ACT 'Peter 'went] ∧ ['DIR1 'C 'went] ∧ ['
    ↪ DIR2 'X 'went] ∧
    [['DIR3 'D 'went] ∨ ['DIR3 'D 'walking]]]]
```

Code 11: The model after application of Path Update

6. Conclusion

In this paper, we described the TIL-Script construction processing used for two purposes.

In the first part, we have described the method of finding an appropriate concept based on properties and attributes' values known by the user. The method exploits Formal Conceptual Analysis applied on the explications of atomic concepts contained in textual sources. The method offers appropriate concepts which fall under properties and attributes' values provided by the user.

In the second part, we described heuristics that obtain descriptions of agent's journeys from descriptions in natural language. Such a description can be used as a navigation tool in complex multi-agent systems environments.

Both methods in the first and second parts of this paper are based on the same supervised machine learning algorithm that processes TIL-Script constructions as examples. The algorithm is adjusted for its purpose.

Acknowledgements

This research has been supported by Grant of SGS No. SP2022/123, VŠB - Technical University of Ostrava, Czech Republic, "Application of Formal Methods in Knowledge Modelling and Software Engineering V" and by CZ.02.2.69/0.0/0.0/18_054/0014696 Development of R&D capacities of the Silesian University in Opava and also supported under the Student Funding Scheme, project SGS/8/2022.

References

[1] Duží, M., Jespersen, B., Materna, P. (2010): Procedural Semantics for Hyperintensional Logic. *Foundations and Applications of Transparent Intensional Logic*. Berlin: Springer.

[2] Menšík, M., Duží, M., Albert, A., Patschka, V., Pajr, M., "Machine learning using TIL". In Frontiers in Artificial Intelligence and Applications, Amsterdam: IOS Press, Vol. 321, pp. 344-362, DOI: 10.3233/FAIA200024

[3] Menšík, M., Duží, M., Albert, A., Patschka, V., Pajr, M. (2019): Refining concepts by machine learning.*Computación y Sistemas*, Vol. 23, No. 3, 2019, pp. 943–958, doi: 10.13053/CyS-23-3-3242

[4] Menšík, M., Duží, M., Albert, A., Patschka, V., Pajr, M. (2019): Seeking relevant information sources. In Informatics'2019, *IEEE 15th International Scientific Conference on In-formatics*, Poprad, Slovakia, pp. 271-276.

[5] Albert, A., Duží, M., Menšík, M., Pajr, M., Patschka, V. (2021): Search for Appropriate Textual Information Sources. In *Frontiers in Artificial Intelligence and Applications*, vol. 333: Information Modelling and Knowledge Bases XXXII, B. Thalheim, M. Tropmann-Frick, H. Jaakkola,N. Yoshida, Y. Kiyoki (eds.), pp. 227-246, Amsterdam: IOS Press, doi: 10.3233/FAIA200832

[6] Menšík, M., Albert, A., Patschka, V. (2020): Using FCA for Seeking Relevant Information Source. In *RASLAN 2020*, Brno: Tribun EU, 2020, 144 p. ISBN 978-80-263-1600-8, ISSN 2336-4289.

[7] Menšík, M., Albert, A., Patschka, V. a Pajr, M. Improvement of Searching for Appropriate Textual Information Sources Using Association Rules and FCA. In: *Frontiers in Artificial Intelligence and Applications*, vol. 343: Information Modelling and Knowledge Bases XXXIII. Amsterdam: IOS Press, 2022, 2022-01-14. Frontiers in Artificial Intelligence and Applications. ISBN 9781643682426. ISSN 978-1-64368-242-6. doi:10.3233/FAIA210487

[8] Menšík M., Albert A., Michalovský T. Using FCA and Concept Explications for Finding an Appropriate Concept. In: Proceedings of the Fifteenth Workshop on Recent Advances in Slavonic Natural Languages Processing, RASLAN 2021. Brno: Tribun EU, 2021, s. 49-60. ISBN 978-80-263-1670-1. ISSN 2336-4289.

[9] Menšík, M., Albert A., Rapant P., Michalovský T. Heuristics for Spatial Data Descriptions ina Multi-agent System. In: To appear.

[10] Tichý P. "Intension in Terms of Turing Machines". In: Studia Logica: An International Journal for Symbolic Logic 24 (1969), str. 7. URL: http://www.jstor.org/stable/20014527.

[11] Tichý P. The Foundations of Frege's Logic. Berlin: De Gruyter, 1988. ISBN: 9783110849264. DOI: 10.1515/9783110849264.

[12] M. Duží, B., Jespersen, P. Materna, "Procedural Semantics for Hyperintensional Logic. Foundations and Applications of Transparent Intensional Logic.", Berlin: Springer, 2010.

[13] Ciprich N., Duzi M., Košinár M. (2009). The TIL-script language. 190. str. 166-179, doi: 10.3233/978-1-58603-957-8-166.

[14] Winston P. H.(1992): Artificial intelligence. 3rd ed.,

Mass.: Addison-Wesley Pub. Co., 1992. ISBN 02-015-3377-4.

[15] Ganter, B., Wille, R.: "Formal Concept Analysis: Mathematical Foundations". 1st ed., Berlin: Springer, 1999, ISBN 978-3-540-62771-5.

[16] Fischer, K., Ágel, V. Dependency grammar and valency theory. In: The Oxford handbook of linguistic analysis; 2010, p. 223-255.