Automated Generation of Control Concepts Annotation Rules Using Inductive Logic Programming

System Description

Basel Shbita[1](✉) and Abha Moitra[2](✉)

[1] University of Southern California, Los Angeles, CA, USA
shbita@usc.edu
[2] General Electric Research, Niskayuna, NY, USA
moitraa@ge.com

Abstract. Capturing domain knowledge is a time-consuming procedure that usually requires the collaboration of a Subject Matter Expert (SME) and a modeling expert to encode the knowledge. This situation is further exacerbated in some domains and applications. The SME may find it challenging to articulate the domain knowledge as a procedure or a set of rules but may find it easier to classify instance data. In the cyber-physical domain, inferring the implemented mathematical concepts in the source code or a different form of representation, such as the Resource Description Framework (RDF), is difficult for the SME, requiring particular expertise in low-level programming or knowledge in Semantic Web technologies. To facilitate this knowledge elicitation from SMEs, we developed a system that automatically generates classification and annotation rules for control concepts in cyber-physical systems (CPS). Our proposed approach leverages the RDF representation of CPS source code and generates the rules using Inductive Logic Programming and semantic technologies. The resulting rules require a small set of labeled instance data that is provided interactively by the SME through a user interface within our system. The generated rules can be inspected, iterated and manually refined.

Keywords: Knowledge capture · Semantic model · Knowledge graphs · Rules · Rule annotation · Cyber-physical systems

1 Introduction

Capturing domain knowledge is a critical task in many domains and applications. This process may involve knowledge elicitation followed by knowledge representation to facilitate inferencing, reasoning, or integration in some decision support systems. Knowledge capture frequently poses a roadblock in developing and

B. Shbita—This work was done while the author was at GE Global Research.

© Springer Nature Switzerland AG 2022
M. Hanus and A. Igarashi (Eds.): FLOPS 2022, LNCS 13215, pp. 171–185, 2022.
https://doi.org/10.1007/978-3-030-99461-7_10

deploying systems that automate processing or reasoning tasks. For instance, a Subject Matter Expert (SME) might have deep domain knowledge but may not be able to describe it in terms of concepts and relationships that can be used to represent the knowledge. Also, at times, the SME may not describe the knowledge at the right level of detail that may be needed for making automated decisions. For example, in the cyber-physical domain, we are often required to analyze a legacy product without an adequate description of its software, imposing a challenge on the system operator. In the same domain, we frequently require the recovery of mathematical structures implemented without having the required software proficiency. Considering the vast amount of code presently used in such systems, the problem becomes intractable and tedious even for an SME and is often susceptible to human error.

To overcome such issues, we use an Inductive Logic Programming (ILP) based approach wherein the SME identifies positive and negative examples for describing a concept in the code. The ILP system uses these examples to derive logic programming rules (i.e., annotation or classification rules) for formally defining those concepts. This allows the SME to quickly detect the desired software modules in his task and route the software's relevant components to designated experts. Our approach is iterative in that the SME can refine the rules learned by adding more positive and negative examples.

Our approach is based on a semantic model (consisting of an ontology and rules), which first describes the basic concepts and relationships of the domain. In order to learn the definition of more complex concepts, the SME provides positive and negative examples that are automatically translated into a formal representation using the basic semantic concepts and relationships.

Our approach tackles both issues described above. Since we automatically translate the example data provided by the SME into a logical representation, the SME does not need to have an understanding of the concepts and relationships in the ontology. In addition, since the SME only identifies positive and negative examples and repeats the learning approach until the acquired knowledge is satisfactory, it provides a way to derive complete and accurate descriptions of the concepts in the domain.

1.1 Inductive Logic Programming

Inductive Logic Programming (ILP) [8] is a branch of Machine Learning that deals with learning theories in logic programs where there is a uniform representation for examples, background knowledge, and hypotheses.

Given background knowledge (B) in the form of a logic program, and positive and negative examples as conjunctions $E+$ and $E-$ of positive and negative literals respectively, an ILP system derives a logic program H such that:

- All the examples in $E+$ can be logically derived from $B \wedge H$, and
- No negative example in $E-$ can be logically derived from $B \wedge H$.

ILP has been successfully used in applications such as Bioinformatics and Natural Language Processing [1,2,5]. A number of ILP implementations are

available. In our work we use Aleph [10] and run it using SWI-Prolog [11]. Aleph bounds the hypothesis space from the most general hypothesis to the most specific one. It starts the search from the most general hypothesis and specialises it by adding literals from the bottom clause until it finds the best hypothesis.

Aleph requires three files to construct theories:

- Text file with a `.b` extension containing background knowledge in the form of logic clauses that encode information relevant to the domain and the instance data.
- Text file with a `.f` extension that includes the positive ground facts of the concept to be learned.
- Text file with a `.n` extension that includes the negative ground facts of the concept to be learned.

As an example, consider an ILP task with the following sets of background knowledge, positive examples, and negative examples:

$$B = \left\{ \begin{array}{c} \texttt{builder(alice).} \\ \texttt{builder(bob).} \\ \texttt{enjoys_lego(alice).} \\ \texttt{enjoys_lego(claire).} \end{array} \right\} E^{+} = \left\{ \texttt{happy(alice).} \right\} E^{-} = \left\{ \begin{array}{c} \texttt{happy(bob).} \\ \texttt{happy(claire).} \end{array} \right\}$$

Given these three sets, ILP induces the following hypothesis (written in reverse implication form, as usually done in logic programs):

$$H = \left\{ \texttt{happy(X) :- builder(X),enjoys_lego(X).} \right\}$$

This hypothesis contains one rule that says: if `builder(X)` and `enjoys_lego(X)` are true, then `happy(X)` must also be true for all `X`. In other words, this rule says that if persons are builders and enjoy lego then they are happy. Having induced a rule, we can deduce knowledge from it. Cropper et al. [4] provides a comprehensive survey of ILP along with various applications.

1.2 Cyber-Physical Systems

Cyber-Physical System (CPS) is a term describing a broad range of complex, multi-disciplinary, physically-aware engineered systems that integrate embedded computing technologies and software into the physical world. The control aspect of the physical phenomena and the theory behind control systems are the basis for all state-of-the-art continuous time dynamical systems and thus have a crucial role in CPS design. In control theory, SMEs describe a system using a set of primitive and higher-level concepts that represent and govern the system's signals and enforce its desired behavior. Table 1 presents several math primitives and higher-level control concepts that are widely used in industrial control systems and a variety of other applications requiring continuously modulated control (e.g., water systems, robotics systems).

Table 1. A partial list of control concepts and math primitives employed in control mechanisms in cyber-physical systems and their description

Concept	Description and associated properties
Constant	Variables that are initialized and not updated
Reference signal	Variable/signal that represent desired behavior or setpoint
Output signal	Variable/signal that goes as output from a control block
Difference	Control block generating a subtraction of two signals
Sum	Control block generating an addition of two signals
Error signal	Variable/signal that is an output of a difference operation with a reference and measured (output) signal
Gain	Control block generating a multiplication between signals and/or scalars
Division	Control block generating a division between signals and/or scalars (often includes some divide by zero protection)
Switch	Control block that selects an input to be an output based on a condition or discrete/boolean input
Magnitude saturation	Control block that limits the input signal to the upper and lower saturation values, where the limit values are pre-defined constants
PI controller	Control block that continuously calculates an error signal and applies a correction based on proportional and integral terms
PID controller	Control block that continuously calculates an error signal and applies a correction based on proportional, integral, and derivative terms

Conventionally, control-policy software are completely separate from the system infrastructure and implemented after manufacturing the system prototype. This presents a challenge for SMEs who are proficient with the required mathematical knowledge and control theory background but are not equipped with a sufficient knowledge in low-level programming or software design. Locating the appropriate code blocks that correspond to a specific control concept (e.g., an integrator) that is of interest to the SME, either for the purpose of validation or reverse engineering, imposes a significant challenge. It is extremely difficult for the SME to recover mathematical structures implemented in the software. Further, the SME can find it challenging to articulate their domain knowledge in a form of code or the formalism required to address their task.

1.3 From Source Code to a Knowledge Graph

Knowledge Graphs (KGs), in the form of RDF statements, are the appropriate representations to store and link complex data. KGs combine expressivity,

interoperability, and standardization in the Semantic Web stack, thus providing a strong foundation for querying and analysis.

The RDF representation of the desired CPS system source code is obtained in two steps. First, from the source code a JSON file is extracted with the method of Pyarelal et al. [9] to describe the function networks and expression trees (i.e., representations of arithmetic expressions). Next, the RDF data is produced by generating RDF triples following a pre-defined semantic model using the materialized JSON instance data. Listing 1.1 shows a portion of the pre-defined semantic model that is used to model the instance data into RDF, expressed in SADL [3] (Semantic Application Design Language). SADL is an open-sourced domain-independent language that provides a formal yet easily understandable representation of models. The SADL tool, which is available as a plugin to Eclipse[1], automatically translates statements in SADL to a Web Ontology Language (OWL) [6] file, which contains the RDF statements (i.e., triples). For example, in our model (as shown in Listing 1.1), a `HyperEdge` is related to a single `Function` via the property `function`, and has multiple `input` and `output` of type `Variable`. It is important to note that this pre-defined model does not vary and is similar for any given code input; it is merely required to define the ontological elements needed to describe code in an RDF form.

Finally, the resulting RDF contains representations of basic code elements found in a given source code. Listing 1.2 shows an excerpt from the resulting knowledge graph of the file `simple_PI_controller.c` shown in Appendix A, expressed in SADL as well.

```
1  HyperEdge is a type of Node
2      described by inputs with values of type Variable
3      described by function with a single value of type Function
4      described by outputs with values of type Variable.
5  Function is a type of Node
6      described by ftype with a single value of type string
7      described by lambda with a single value of type string
8      described by expression_tree with a single value of type ExpressionTree.
9  Variable is a type of ExpNode
10     described by object_ref with a single value of type string
11     described by data_type with a single value of type string.
```

Listing 1.1. A portion of the semantic model used to model the data (written in SADL). A `Function` is a `Node` with the relations `ftype` (function type) and `lambda` with a range of type `string` and an `expression_tree` of type `ExpressionTree`

```
1  _21df3f15-1763-9632-e936-8aca2281a699 is a grfnem:Function,
2     has grfnem:metadata (a grfnem:Metadata with grfnem:line_begin 12),
3     has grfnem:ftype "ASSIGN",
4     has grfnem:lambda "lambda error,Kp_M,Ki_M,integrator_state: ((error * Kp_M) + (Ki_M *
          ↪ integrator_state))".
5  d4000e07-fe4a-aa23-b882-1030d655eee0 is a grfnem:Variable,
6     has grfnem:metadata (a grfnem:Metadata with grfnem:line_begin 27, with grfnem:from_source true),
7     has grfnem:identifier "simple_PI_controller::simple_PI_controller.main::Kp_M::0".
```

Listing 1.2. An excerpt from the resulting knowledge graph representation (written in SADL) of the file `simple_PI_controller.c` (see Appendix A). The `Function` shown above is of type `ASSIGN` and starts at line 12. This is the resulting instance data that is automatically generated from the `c` source code

[1] https://www.eclipse.org/.

2 Integrated Control-Concept Induction Platform

The question we are addressing is *how can we streamline and leverage the CPS program knowledge graph data to capture domain-knowledge and assist the SME with additional knowledge discovery?* Knowledge discovery in data is the non-trivial extraction of implicit, previously unknown, and potentially useful information from data.

As we mentioned earlier, and in order to overcome this challenge, we use an ILP-based approach wherein the SME essentially identifies positive and negative examples for describing a concept. The ILP system uses them to derive logic programming rules for formally defining that concept.

2.1 Problem Definition

The task we address here is as follows: Given an input in the form of an OWL file containing RDF triples that represent basic code elements found in the source code (as seen in Listing 1.2 in SADL format), we want to generate logic programming rules for formally defining control concepts and math primitives (e.g., a Constant, see Table 1) that are provided as example instances interactively by the SME. The rules should be expressed with Horn logic clauses, similarly to the example we have shown in Sect. 1.1. We require the solution to be iterative in the sense that the SME can refine the learned rules by adding more positive and negative examples.

2.2 Overview of Our Approach: An ILP Platform

As described in Sect. 1.3, the knowledge graph, constructed from the function networks and expression trees, is materialized in an OWL format. Our suggested platform and approach consists of several steps and components, as illustrated in Fig. 1. The platform consists of a module (`owl2aleph`) that automatically translates the OWL data into background knowledge (clauses and instances), namely B, in a format required by Aleph, the ILP system. The module then invokes an interactive user interface in which the SME selects positive and negative instances, namely $E+$ and $E-$ respectively. Lastly, Aleph is invoked using SWI-Prolog to produce a hypothesized clause H (i.e., the learned rule) and provide a list of new positive instances that adhere to H, so that the SME can evaluate the accuracy of the learned rule and select new examples to refine it.

Since we automatically translate the example data provided by the SME into a logical representation (i.e., "Aleph format"), the SME is not required to have knowledge of the concepts and relationships in the ontology. Also, since the SME only identifies positive and negative examples and repeats the learning approach until the knowledge learned is satisfactory, it provides a convenient and fast way for deriving complete and accurate descriptions of the concepts in the domain. The iterative nature of the approach is illustrated via the loop seen in the lower right side of Fig. 1. The loop runs through the SME (i.e., User), the Examples Selection UI (producing $E+$ and $E-$), then through Aleph to produce a new learned rule (i.e., H) then back to the SME.

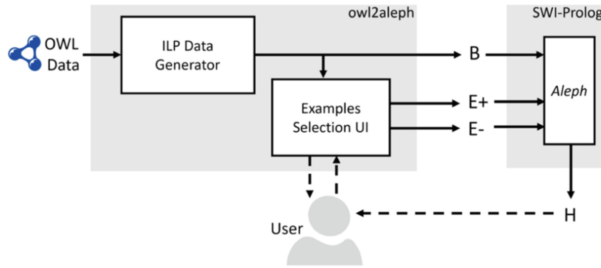

Fig. 1. The integrated control-concept induction platform for deriving math primitives and control concepts classification rules

2.3 Generating the ILP Data

The `owl2aleph` module, which generates the ILP data, consists of two main components (as seen previously in Fig. 1). The architecture of the module is detailed in Fig. 2. We construct the ILP instances based on hyperedges, function nodes, variable nodes, and expression trees based on the semantic model that describes the basic concepts and relationships of the domain (Listing 1.1) and executed over the instance data (Listing 1.2).

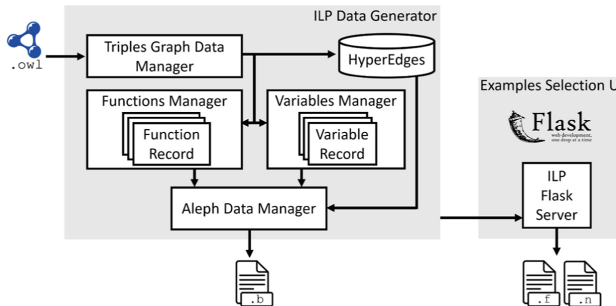

Fig. 2. Architecture of the `owl2aleph` module, generator of the ILP data files

As seen in Fig. 2, there are several sub-components, each one is designed to tackle a different task:

- **Triples Graph Data Manager** reads the OWL data and provides an easy serialization functionality over the given RDF statements. The manager classifies each statement by their functionality to serve other components (i.e., Functions Manager, Variables Manager, HyperEdges).
- **Variables Manager** performs variables disambiguation to enable linking variable nodes (explicit and implicit variables in the source code) and is also responsible for generating attributes regarding their usage (assignments, updates, usage inside other blocks, etc...).

- **Functions Manager** generates the relevant information about the code statement (hyperedge) functionality, the arithmetic operations (expression tree attributes such as multiplication, division, etc...) present in the code statement, and block level attributes (e.g., in a loop call).
- **HyperEdges** is a database of hyperedges, each one represents a code statement. Each hyperedge corresponds to zero or more variable nodes and a single function node. Hyperedges are the primary instances we use to aggregate the information about the math primitives and the control concepts we would like to form logic programs about. In the final pre-inference stage (upon selection of the positive and negative examples), a hyperedge is named as "`newfeature`" for each example, so that the same process and structure can be used for generating rules for any given concept.
- **Aleph Data Manager** generates the background knowledge in an Aleph format (`.b` file). This also includes the construction of definite clauses and additional constraints (rules about predicates and their inputs and outputs) from the source code and materializes the instance data.
- **ILP Flask Server** Runs a Flask[2] application (local HTTP server) to enable an interaction with the SME (User) via a web browser. The application provides a user-friendly interface to inspect and select the hyperedge instances that are positive and negative and to generate them in the required Aleph format (`.f` and `.n` files).

```
10 :- modeh(1,newfeature(+hyperedge)).

30 :- modeb(*,outputs_of(+xvariable,-hyperedge)).
31 :- modeb(*,xfunction(+hyperedge,-xnode)).
32 :- modeb(*,has_operator_mult(+xnode)).
33 :- modeb(*,has_operator_ifexpr(+xnode)).
34 :- modeb(*,xinterface(+xnode)).
35 :- modeb(*,var_assigned_once(+xvariable)).
36 :- modeb(*,var_multi_assigned(+xvariable)).

50 :- determination(newfeature/1,outputs/2).
51 :- determination(newfeature/1,xliteral/1).
52 :- determination(newfeature/1,xfunction/2).
53 :- determination(newfeature/1,inputs/2).
54 :- determination(newfeature/1,xassign/1).

86 hyperedge(xa1f96097abb845469e519fb7f237f9f9).
87 outputs(xa1f96097abb845469e519fb7f237f9f9,variable3).
88 xnode(node2).
89 xassign(node2).
90 has_operator_div(node2).
91 xfunction(xa1f96097abb845469e519fb7f237f9f9,node2).
```

Fig. 3. An excerpt of a resulting background knowledge (`.b`) file

In Fig. 3, we show an excerpt from a background knowledge (`.b`) file. The file includes four different sections of encoded knowledge. First, the `modeh` clause

[2] https://flask.palletsprojects.com/.

defines the hypothesis and takes a `hyperedge` as an input (highlighted with yellow). Second, the `modeb` clauses define the signatures of the predicate functions. Third, the `determination` clauses specify what concepts can be used in rules and how many arguments each one takes. The last section includes the instance data describing the entire CPS program in a logic formalism. All of the above is generated automatically from the RDF data.

The example files (`.f`/`.n`) we provide as input to Aleph simply list a collection of positive or negative `hyperedge`s that correspond to the desired concept we want to learn. These files are generated automatically using the Examples Selection User Interface by simply inspecting their attributes and then adding them either as positives or negatives.

A snapshot of the user interface (UI) is shown in Fig. 4. It is fairly straightforward to operate the UI. The system lists all available `hyperedge`s with their relevant information (line numbers, functionality types, etc...). The user can add a `hyperedge` as a positive or negative example or simply ignore it. Once ready, the user can generate the example files to trigger the next step in the pipeline.

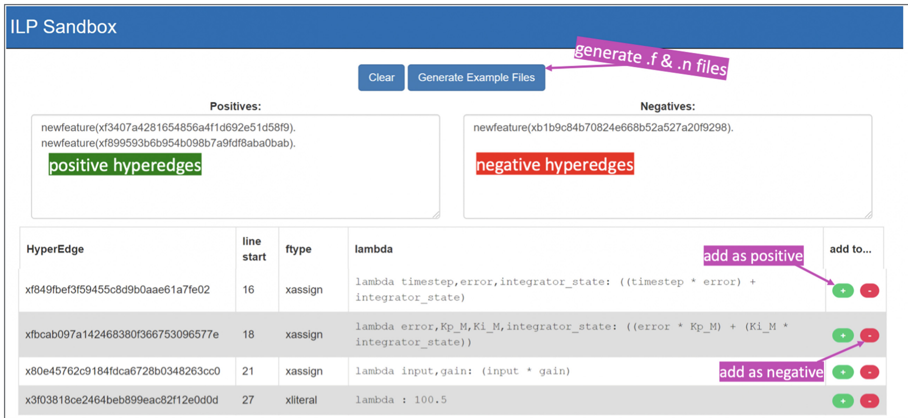

Fig. 4. The examples selection user interface for generating Aleph example files

By automating the translation and modeling of the semantic data into ILP rules, clauses, and instances, and by enabling a straightforward process of example files creation, we can quickly generate classification rules for formally identifying math primitives and control concepts in an iterative, fast, and interactive fashion.

2.4 Rule Generation from ILP Data via an Illustrative Example

Given the ILP data (B, $E+$, $E-$) in Aleph format, we can now trigger the execution of the ILP platform using SWI-Prolog to induce the learned rule, i.e., the hypothesis (H). The outcome is classification rules, expressed in domain

terms, for formally identifying math primitives and control concepts. The SME identifies positive and negative examples and repeats the learning approach until the knowledge learned rule is satisfactory.

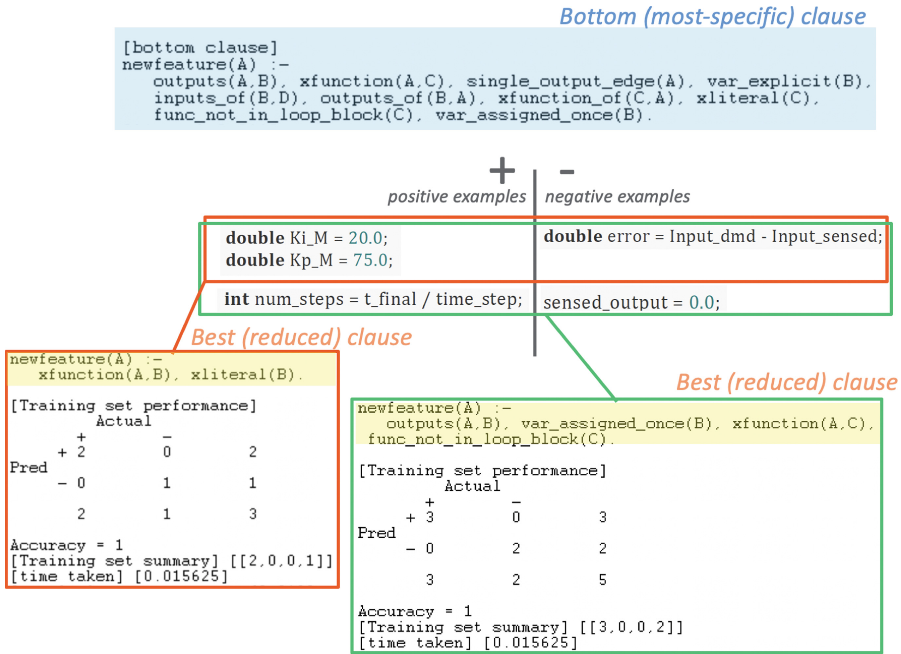

Fig. 5. An illustrative example of the learning of the control concept of "Constant" (Color figure online)

In Fig. 5, we illustrate the working of the developed ILP infrastructure to learn a simple classification rule to identify the mathematical concept of "Constant" (i.e., an expression with a variable assignment that is initialized and not updated in the code). The selected examples in Fig. 5, and the ones discussed in this section, correspond to code statements in the file `simple_PI_controller.c` shown in Appendix A. In the first iteration in this scenario, the SME selects two positive examples and a single negative example (upper orange box in Fig. 5) corresponding to the code statements in lines 26 and 27 as positives, and line 8 as a negative. As explained in Sect. 1.1, the ILP system constructs the most specific clause (given B and entailing $E+$), which is shown highlighted in blue. The generated rule in this execution produces the rule (also seen highlighted in yellow inside the lower orange box):

```
newfeature(A) :- xfunction(A,B), xliteral(B).
```

Which basically means that `hyperedge A` corresponds to a "Constant" (the `newfeature`) if `A` has a function `B` that is an assignment to a literal.

Upon query of the instances that adhere to the generated rule, the SME can add more examples, either as a positive or a negative. The SME then selects an additional positive and negative example in a second iteration (upper green box in Fig. 5) corresponding to the code statement in line 29 as a positive, and line 38 as a negative. The generated rule in this execution produces the rule (also seen highlighted in yellow inside the lower green box):

```
newfeature(A) :- outputs(A,B), var_assigned_once(B),
        xfunction(A,C), func_not_in_loop_block(C).
```

Which means that `hyperedge A` corresponds to a "Constant" if `A` has a variable `B` that is assigned only once and the `hyperedge A` has a function `C` in which the assignment is not inside any loop, as we would have expected.

The ILP infrastructure enables an automatic, iterative, and fast process for capturing domain knowledge for math primitives and control concepts in the form of classification rules. The resulting rules are used as feedback for the SME and can be further utilized to learn additional levels of knowledge.

3 Evaluation and Discussion

We evaluate the ILP-based approach for learning classification rules for control and math primitives on a dataset consisting of three OWL files originating from three source code files driving proportional-integral (PI) controllers with a simple plant model. The dataset consists of 8974 triples pertaining to 61 different instances of math and control concepts.

We have been successful in generating classification rules for simple math primitives and several control concepts. Table 2 shows a summary of the results. For each concept, we show the size of the training data (number of positive and negative examples provided), the number of bottom clause literals before the learning, the total number of reduced clause literals after the learning, the learning time, number of true positives, number of false positives, number of false negatives, precision, recall, and the F1 score. We note that the concepts of "Switch", "Magnitude saturation", and "PID controller" could not be learned due to an insufficient number of positive examples. We require at least two positive examples per concept to generate the most specific clause that initiates the ILP process.

As seen in Table 2, for 7 out of the applicable 9 concepts, the resulting generated rules had a perfect F1 score (maximum precision and recall) and a significant reduction in the number of literals in the generated rule (from bottom clause, pre-learning, to reduced clause, post-learning). Additionally, the process took less than a second to complete for all concepts shown in the table, which is a crucial and important ability to have in such a problem setting. Further, it required no more than two positive examples (and no more than 5 negatives, depending on the complexity of the concept in our data) to generate the final rule for all concepts.

Table 2. Results summary for the ILP generated rules for our targeted math and control concepts

Concept	(E+, E−)	Bottom clause size	Reduced clause size	Time [seconds]	True positives	False positives	False negatives	Precision	Recall	F1
Difference	(2, 1)	24	3	0.063	4	0	0	1.0	1.0	**1.0**
Sum	(2, 2)	29	3	0.063	9	0	0	1.0	1.0	**1.0**
Gain	(2, 1)	23	3	0.047	9	0	0	1.0	1.0	**1.0**
Division	(2, 2)	24	3	0.094	3	0	0	1.0	1.0	**1.0**
Constant	(2, 5)	11	6	0.125	23	0	0	1.0	1.0	**1.0**
Error signal	(2, 5)	24	6	0.859	2	0	0	1.0	1.0	**1.0**
PI controller	(2, 5)	45	5	0.453	3	0	0	1.0	1.0	**1.0**
Output signal	(2, 3)	12	12	0.859	3	3	0	0.50	1.0	0.67
Reference signal	(2, 2)	11	11	0.375	3	17	0	0.15	1.0	0.26
Switch	Not enough positives ($E+$)									
Magnitude saturation										
PID controller										

One must note that the number of iterations needed is dependent on how many and which positive and negative examples are selected by the SME. For example, suppose a user chooses similar positive or negative examples. In that case, it could be not very meaningful in converging towards a more reduced clause, requiring the user to pick additional and substantially different examples.

The remaining two concepts out of the applicable 9 concepts ("Output Signal" and "Reference Signal") did not achieve a high F1 score (or any reduction in the size of the clause literals) since there is not enough data to separate the positive examples from the negative examples in these concepts. The training data must have a sufficient number of positive examples with a certain amount of "richness" (diversity in implementation and usage) to enable the separation of the examples to generate an accurate and satisfying rule. This is an expected requirement, as there are some concepts that can be coded in different approaches (logic vs. arithmetic). Further, the code can have several mutations even if implemented using the same "approach". For example, the code could include pointers that get allocated dynamically, imposing a difficulty in our approach, which relies on a static semantic analysis.

One must note that these scores reflect the accuracy of the rules within our dataset. The same rules, if executed on a different dataset, may not necessarily produce similar results. We inspected the generated rules with the prefect F1 scores. We noticed that some of the rules are aligned with our expectations. For example, the generated rule for the concept of "Gain" was:

```
gain(A) :- xfunction(A,B), has_operator_mult(B).
```

Which means the `hyperedge A` has a function `B` that includes the multiplication operator, as expected.

Other rules were not completely aligned to what we would expect the SME to define. For example, the generated rule for the concept of "PI Controller" was:

```
picontroller(A) :- outputs(A,B), var_implicit(B),
        xfunction(A,C), has_operator_add(C).
```

Which is not sufficient to capture the two control terms of proportional and integral operations of addition and multiplication that are required to define a PI controller, but it was sufficient to capture the 3 instances that exist in our dataset accurately.

The quality of the ILP generated rules is dependent on the supplied input. Sub-par rules result from inadequate input data that does not hold sufficient information about the targeted concept. By providing additional code examples and richer data, we provide better coverage and generate more accurate rules.

4 Related Work and Conclusions

Since a vast amount of domain knowledge has already been captured in text, considerable effort has been made in extracting this written knowledge into formal models. Wong et al. [12] provides a survey of various approaches. Most of this effort has been in extracting concepts and relationships between the concepts and representing it in a semantic model. We have also previously used ILP in the domain of Design for Manufacturability (DFM) where the goal was to design products that are easier to manufacture by providing early manufacturability feedback in Moitra et al. [7].

In this work we have considered how we can automate the capture of Cyber-Physical Systems (CPS) domain knowledge by applying Inductive Logic Programming (ILP) to positive and negative instance data in RDF format, originating from a CPS program source code. We have shown this by developing an Integrated Control-Concept Induction Platform for generating annotation and classification rules for control concepts and math primitives. Our approach is feasible and effective in terms of time, completeness, and robustness. These early results we have shown are encouraging and provide promising opportunities and applications.

Acknowledgements. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under the Agreement No. HR00112190017. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

A simple_PI_controller.c

A PI controller simple_PI_controller.c used in Sect. 2.4 is shown below. The selected examples in Fig. 5 correspond to code statements in this file.

```
#include <stdio.h>

double integrator_state = 0.0;

/* Simple PI controller */
double PI_calc(double Input_dmd, double Input_sensed, double Kp_M, double Ki_M, double
    ↪ timestep)
{
    double error = Input_dmd - Input_sensed; // negative example for ILP (iteration 1)

    integrator_state = integrator_state + timestep*error;

    return error*Kp_M + integrator_state*Ki_M;
}

/* Proportional plant! */
double plant_model(double input, double gain)
{
    return input*gain;
}

int main(int argc, char **argv)
{
    double t_final = 100.5;
    double time_step = 0.015;

    double Ki_M = 20.0; // positive example for ILP (iteration 1)
    double Kp_M = 75.0; // positive example for ILP (iteration 1)

    int num_steps = t_final / time_step; // positive example for ILP (iteration 2)

    double desired_output = 10.0;

    double plant_command;
    double sensed_output;

    double plant_gain = 0.01;

    sensed_output = 0.0; // negative example for ILP (iteration 2)

    for (int i = 0; i < num_steps; i++)
    {
        plant_command = PI_calc(desired_output, sensed_output, Kp_M, Ki_M, time_step);

        sensed_output = plant_model(plant_command, plant_gain);

        printf("%f, %f, %f", (double)i*time_step, plant_command, sensed_output);
    }

    return 0;
}
```

Listing 1.3. simple_PI_controller.c

References

1. Bratko, I., Muggleton, S.: Applications of inductive logic programming. Commun. ACM **38**(11), 65–70 (1995)
2. Chen, D., Mooney, R.: Learning to interpret natural language navigation instructions from observations. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 25 (2011)
3. Crapo, A., Moitra, A.: Toward a unified English-like representation of semantic models, data, and graph patterns for subject matter experts. Int. J. Semant. Comput. **7**(03), 215–236 (2013)
4. Cropper, A., Dumančić, S., Evans, R., Muggleton, S.H.: Inductive logic programming at 30. Mach. Learn. **111**(1), 147–172 (2022). Springer
5. Faruquie, T.A., Srinivasan, A., King, R.D.: Topic models with relational features for drug design. In: Riguzzi, F., Železný, F. (eds.) ILP 2012. LNCS (LNAI), vol. 7842, pp. 45–57. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38812-5_4
6. McGuinness, D.L., Van Harmelen, F., et al.: Owl web ontology language overview. W3C Recommend. **10**(10), 2004 (2004)
7. Moitra, A., Palla, R., Rangarajan, A.: Automated capture and execution of manufacturability rules using inductive logic programming. In: Twenty-Eighth IAAI Conference (2016)
8. Muggleton, S.: Inductive logic programming. New Gener. Comput. **8**(4), 295–318 (1991)
9. Pyarelal, A., et al.: Automates: automated model assembly from text, equations, and software. arXiv preprint arXiv:2001.07295 (2020)
10. Srinivasan, A.: The aleph manual (2001)
11. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-prolog. Theory Pract. Logic Program. **12**(1–2), 67–96 (2012)
12. Wong, W., Liu, W., Bennamoun, M.: Ontology learning from text: a look back and into the future. ACM Comput. Surv. (CSUR) **44**(4), 1–36 (2012)