Interactive NLU-Powered Ontology-Based Workflow Synthesis for FAIR Support of HPC

Zifan Nan*, Mithil Dave*, Xipeng Shen*, Chunhua Liao† Tristan Vanderbruggen†,
Pei-Hung Lin†, Murali Emani‡
{znan,mdave2,xshen5}@ncsu.edu, {liao6,Vanderbruggen1,lin32}@llnl.gov, memani@anl.gov,
*Department of Computer Science, North Carolina State University, Raleigh, NC, USA
†Lawrence Livermore National Laboratory, Livermore, CA, USA
‡Argonne National Laboratory, Lemont, IL, USA

***Abstract*—Workflow synthesis is important for automatically creating the data processing workflow in a FAIR data management system for HPC. Previous methods are table-based, rigid and not scalable. This paper addresses these limitations by developing a new approach to workflow synthesis, *interactive NLU-powered ontology-based workflow synthesis (INPOWS)*. INPOWS allows the use of Natural Language for queries, maximizes the robustness in handling concepts and language ambiguities through an interactive ontology-based design, and achieves superior extensibility by adopting a synthesis algorithm powered by Natural Language Understanding. In our experiments, INPOWS shows the efficacy in enabling flexible, robust, and extensible workflow synthesis.**

***Index Terms*—Ontology, Workflow, Synthesis, HPC, FAIR, NLP**

I. Introduction

Workflow synthesis is a technology that assembles a number of processing units (e.g., scripts, commands, APIs) into an executable, the execution of which produces results requested by an input query. Workflow synthesis plays an important role in improving productivity, especially for scientific domains that exhibit a high complexity in data and sets of operations. Example use of workflow synthesis exists in remote sensing [61], mass spectrometry-based proteomics [38], and many other domains.

Workflow synthesis is becoming more important for HPC as well, partially driven by the rapid growth of HPC data and the need for better ways to share and (re)use them in HPC. HPC contains many kinds of tasks and deals with a large variety of data. Recent years have witnessed an increasing interest in building up FAIR (Findable, Accessible, Interoperable, Reproducible) repositories so that the data can be easily shared and reused. An example is a recently proposed framework, HPC-FAIR [55]. As many datasets are uploaded by different users into the repository, it is essential to have a way to help users make sense of the data in the repository and quickly put together a workflow that can retrieve the relevant datasets, reformat them when necessary, process them, and extract out the finally useful elements. Automatic workflow synthesis is a solution for meeting the needs.

There have been some efforts trying to automate workflow synthesis (or composition). The topic of automated synthesis and planning of workflows has been discussed at least since the 1990s [29], [45], [50], and gained new attention in the early 2000s when (semantic) web services became popular [1], [46]. Example frameworks include the semantic service composition approaches in myGrid [4], [27], agent-based approaches [30], OWL-based SADI framework with its SHARE client for web service pipelining [57], and the PROPHETS framework that makes use of temporal-logic synthesis [19]–[21], [36].

All the prior approaches depend on rich, consistent component annotations in precise terms. In the work on mass spectrometry-based proteomics [38], for instance, both user's intents and the annotations of the inputs, outputs, operations of every building block (software tools) of the workflows in the domain must be written in EDAM, a standard set of terms created by the Bioinformatics community that defines the topics, operations, types of data and data identifiers, and data formats, relevant in data analysis and data management in life sciences.

The prior approach works for a domain that is stable with a limited set of concepts and types of entities, but does not work for HPC. HPC is a loose defined domain, with a multitude of types of data, many and continuously expanding operations, and unlimited number of possible tasks. Even if it is possible to create a fixed vocabulary to capture most commonly used concepts, operations, and types of data objects, it would be large and complex, difficult for users to use consistently.

This work aims to address the problem by exploring a new approach to workflow synthesis. Our proposal is *interactive NLU-powered ontology-based workflow synthesis (INPOWS)*. INPOWS has several distinctive features.

(i) It offers *flexibility*, removing the rigidity of prior methods by allowing the use of Natural Language as its user interface. Users can submit their queries written in free Natural Languages (NL) (currently, English). For general users, NL is the most intuitive way of expression in many domains; such an interface addresses the shortcomings of rigid tables, offering flexibility for users.

(ii) It maximizes the *robustness in handling concepts* in a domain by building the synthesis on Ontology. Ontology refers to a set of concepts and categories in a subject area or domain. It also shows their properties and the relations between them. An ontology of a domain consists of a vocabulary that represents the concepts, categories, properties, and

relations in the domain. It not only offers a way to standardize expressions in the domain, but more importantly, by organizing concepts and their relations formally. In the end, ontology makes automatic reasoning of the concepts in the domain possible. The reasoning enables INPOWS to bridge the gaps between the concepts used in a query and the concepts in datasets.

(iii) INPOWS achieves *robustness in handling NL ambiguities* through an interactive design. NL queries can be ambiguous. INPOWS addresses it by popping up hints and choices when a user inputs her query; it helps clarify the intent of the user and simplifies the synthesis.

(iv) INPOWS achieves superior *extensibility* by adopting a synthesis algorithm powered by Natural Language Understanding (NLU). Unlike common data-driven approaches, the NLU-powered synthesis algorithm does not need a time-consuming training process on hundreds of thousands of labeled examples. It draws on the understanding of the semantic and syntax of the domain by analyzing the domain documentations. By eliminating the requirement of large data collection, this feature eases the building of INPOWS for a domain. More importantly, it makes the domain easily extensible. When a new script is added into a domain, the user just needs to provide a description of the script (usually in one or several sentences); INPOWS can then naturally take that new script into its synthesis process.

We developed INPOWS and integrated it into a FAIR data management system for HPC, named HPC-FAIR. Our experiments show that INPOWS can achieve 80% synthesis accuracy on 60 HPC-FAIR workflow synthesis tasks. Overall, INPOWS makes the following major contributions:

- To the best of our knowledge, INPOWS is the first workflow synthesizer that allows NL queries and at the same time achieves robust results (over 80% accuracy) on complex continuously changing domains.
- INPOWS proposes the first method that seamlessly integrates ontology with NLU-powered workflow synthesis.
- INPOWS develops an interactive design to resolve the difficulties that NL ambiguity imposes on NLU-powered workflow synthesis.
- Evaluated empirically, INPOWS shows 80% synthesis accuracy for its effectiveness.

II. Background

This work uses HPC-FAIR as the basis for its hosting of a wide range of HPC data to meet many kinds of HPC tasks. This section provides some background of HPC-FAIR and code synthesis.

A. HPC-FAIR and HPC Ontology

The FAIR Guiding Principles [58] aim to improve the findability, accessibility, interoperability, and reusability of digital content by making them both human and machine actionable. HPC-FAIR [55] is a framework to implement the FAIR principles for HPC's ML models and datasets. Within this framework, information about HPC machine learning data and models from different sources is collected and stored within a Resource Description Framework (RDF) database.

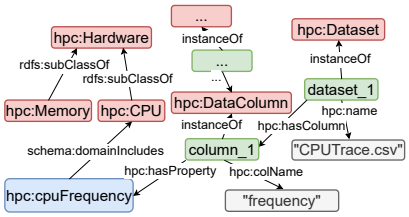

Figure 1. HPC Ontology

Within HPC-FAIR, the HPC ontology [23] serves as a core component to power the RDF database and enable *'Interoperability'* and *'Reusability'* of the training datasets and AI models in the HPC domain. An ontology provides a common vocabulary to represent and share the domain concepts. It is a formal specification for describing the knowledge about properties, relationships and entities in a domain. The HPC ontology captures information in the HPC domain, using a formal knowledge representation of Web Ontology Language (OWL). It provides descriptive, administrative and statistical information about the properties of major concepts in the HPC domain such as computers, projects, AI models, software, hardware, datasets etc.

The ontology is built on a set of guidelines which cater to our requirements to make dataset and AI models FAIR for the community. The ontology design consists of two parts. A high-level ontology to capture core components of the domain such as dataset and AI models. A set of low-level components represent detailed, internal information of various sub domains such as compilers, GPUs, performance data etc. A single namespace with a prefix named 'hpc:' is used to include all concepts and relations under the HPC domain. For example, hpc:Dataset is used to represent the dataset concept while hpc:wasAttributeTo indicates the relation between a dataset and a person.

The information in ontology is represented in an RDF (Resource Description Framework) format of *(subject, property, object)*, which can be visualized as nodes (for subjects and objects) and edges (for properties). Figure 1 shows example information stored in the HPC ontology. For example, *('Memory', 'SubClassof', 'Hardware')* triple states that *"Memory is a sub class of Hardware"*. *dataset_1* named *"CPUTrace.csv"* is an instance of class Dataset. This dataset instance has a column named *"frequency"*. The column is related to the property named *"cpuFrequency"*, which in turn is a CPU related property as encoded using *"schema:domainIncludes"*. It is a part of the HPC-FAIR data uploading process to annotate columns to the associated properties. We use an RDF query language named SPARQL [41] to extract information from ontology to synthesize our workflow.

B. Workflows Synthesis on HPC-FAIR

HPC-FAIR hosts HPC datasets and models uploaded by various users. Thus, the data and models needed by an user may be the assembled result from multiple sources. The workflow

Table I
WORKFLOW QUERY AND EXPRESSION EXAMPLES

ID	Type	Workflow NL query	Workflow expression
1	Data Manipulation	Merge dataset "X.csv" and dataset "Y.json"	MergeDataset(csvFile(string("X.csv")), json2csv(jsonFile(string("Y.json"))))
2	Ontology query	Get datasets whose subject is "GPGPU"	GetDataset(datasetName(string("lassen_overhead, performance_results_dataset")))
3	Combination	Get CPU related columns from dataset "CPUTrace.csv"	GetColumn(columnName(string("flops","frequency")), datasetName(string("CPUTrace.csv")))

synthesis task in HPC-FAIR aims to automatically create a data processing workflow which, from existing datasets, can automatically derive the data and models that meet the needs of a user based on her queries.

The common needs of users on HPC-FAIR include data manipulations and knowledge queries. Table I shows three example workflow synthesis tasks. The data manipulation workflow (Table I(1)) includes tasks such as data merging, data extraction, file type transformation, executing user uploaded scripts, etc. The Ontology workflow (Table I(2)) queries knowledge from the HPC-Ontology based on certain properties, such as the subject of the dataset, the author of a dataset, etc. These two types of workflow can also be combined together to manipulate the data based on the knowledge from Ontology. For example, the workflow query in Table I(3) is used to extract the columns that are related to CPU from a dataset.

To enable the flexibility for users to express their intents, the workflow queries are Natural Language sentences that can express the tasks that are expected to perform by the workflow. For example, Table I(1) indicate the task of merging two dataset with given dataset names. Table I(2) asked for the dataset with specific conditions, which is the subject of the dataset is "GPGPU".

We use workflow expressions to represent the workflows. The execution of workflow expression is the same as function calls in Python, which calls the inner functions first, and the return values of the inner functions become the parameters of the outer functions. For example, the workflow expression in Table I(1) merges two dataset. When executing, the API `csvFile` first reads the file "X.csv". Then the API `jsonFile` reads the file "Y.json", and converts it to *csv* format via API `json2csv`. Finally, the API `MergeDataset` merges two files.

C. NLU-driven Synthesizer

This work builds on the basis of HISyn [32], a Natural Language Understanding (NLU)-driven code synthesizer. Taking the NL query as input, it uses modern NLP techniques to extract key information and their relations inside the query. By comparing the semantics of the key information and the API descriptions of the DSL, it matches the APIs that are related to the key information, and constructs code expressions following the grammar of the DSL. There are three reasons to select HISyn as the workflow synthesizer.

1) NL input: Unlike inputs using a query language or domain specific language, a natural language input allows users to freely express their intents without learning complex grammar or knowledge. HISyn, as a NLU-driven synthesizer, takes NL as an input query and automatically generates the code expression that can fulfill the user's intents.

2) No need for training: Many NL-based synthesizers require examples to train a model to synthesize the codes [5], [24]. The number of training examples varies from thousands to millions. However, as a new domain, HPC-FAIR does not have such a large number of training cases. On the contrary, HISyn employs grammar graph-based translation algorithms, which take the mentioned IR as input, and generate the target code expression without any training.

3) Cross-domain extensibility: HPC-FAIR collects data and models from different sources and projects. Therefore, the workflow synthesizer should be extensible to adept to the growing number of resources. HISyn features cross-domain extensibility. With modular framework architecture of HISyn, it can support the code synthesis tasks from different domains. The HISyn design encapsulates domain-specific elements into separate modules equipped with an easy-to-use interface.

III. CHALLENGES AND SOLUTIONS OVERVIEW

INPOWS features flexible NL inputs, robustness in handling concepts and NL ambiguities, and superior extensibility. It is inspired by HISyn, a NLU-driven code synthesizer. There is yet a large gap between code synthesizer and workflow synthesis for HPC. We list several major challenges in this section.

The first challenge is to create a formal representation of the space of workflows in the domain. As a code synthesizer, HISyn uses domain-specific language (DSL) as the abstraction of the set of possible expressions in a target domain. So the first fold of questions to answer for workflow synthesis via NLU are whether DSL can serve as the right abstraction for the space of workflows and how to represent the scripts and functions inside the synthesizer for reasoning.

The second challenge is on how to robustly handle NL inputs. On one hand, HISyn synthesizes code expressions based on DSL knowledge, hence the synthesis search space is the DSL grammar. On the other hand, INPOWS aims to handle the domain knowledge stored in Ontology, hence the Ontology becomes another search space. Thus, there are two different search spaces in the framework. However, the NL queries are written in free NL, containing information related to both search spaces. Therefore, how to identify the corresponding search space of the key information inside the NL query is another challenge.

The third challenge is on bridging the gaps between the concepts in the user queries and the concept hierarchy in HPC Ontology. As mentioned in Section II-A, SPARQL query language is usually used in searching for information inside the

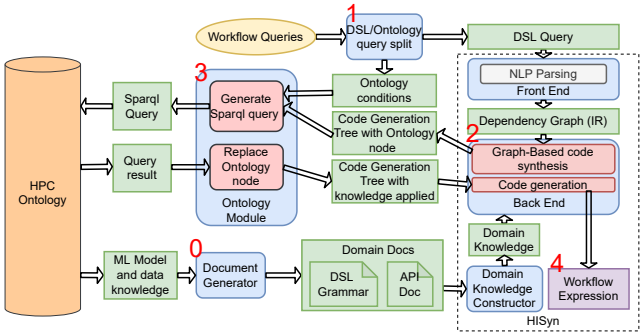

Figure 2. Overall framework of INPOWS

Ontology. To take full advantage of the Ontology, the solution must be able to generate appropriate SPARQL queries from the input queries expressed in NL.

Our exploration addresses these challenges and produces INPOWS, a workflow synthesis framework that generates the workflow expressions which can be directly executed inside a Python script. The overall framework of INPOWS is shown in Figure 2. Besides the HISyn that synthesizes workflow expressions, the framework includes three new modules: Documentation Generator, DSL/Ontology query split, and Ontology Module. These three new modules, together with the whole framework design, solve the above challenges.

Specifically, to address the first challenge, before the workflow synthesis task starts, Document Generator (marked using 0 in Figure 2) is used to prepare the documents needed by HISyn to synthesize the workflow expression. We define a DSL that can include both data manipulation APIs and the operation scripts (building blocks of workflows) uploaded by users. Inside this module, the operation scripts are abstracted into APIs of the DSL. They are added into the DSL grammar and documentation, which are two essential documents for HISyn to synthesize the workflow expression. This module builds the foundation of the workflow synthesis.

Next, when INPOWS receives NL workflow queries, it sends the query to the DSL/Ontology query split (1), which solves the second challenge. The split is an interactive module. The module scans the query to identify the words and phrases that relate to Ontology, and replaces them with corresponding Ontology objects or properties. The queries with replaced Ontology components are then shown to users to determine the one that meets their intents. After that, the correct query is split into two parts. The DSL query is used to synthesize the workflow expression based on the DSL search space. The ontology conditions are used to create a SPARQL query to acquire the knowledge from HPC Ontology.

The Ontology Module (3) takes the Ontology conditions from the split as input, together with the information of the intermediate synthesis results, code generation tree (CGT), from the backend of HISyn (2), to generate the SPARQL query that can search for the corresponding information inside the HPC Ontology. The query results are then used to replace the special nodes inside the CGT to generate the final workflow expression (4). Thus this module resolves the third challenge.

IV. INPOWS

This section explains the design of INPOWS. It first introduces a set of terms and definitions used in the framework, and then explains the document generator and the overall workflow synthesis process with a running example (Figure 3).

A. Prerequisite

There are three data structures/concepts that appear during the synthesis process: the grammar and *grammar graph* that are used by HISyn as the search space, the *dependency graph* that represent the input NL query and serves as an intermediate result (IR) between the front end and back end of HISyn, and the *code generation tree* that represent the synthesis results.

1) Grammar and grammar graph: The context free grammar (CFG) is used to represent the DSL. A CFG is a quadruple $(\mathcal{T}, \mathcal{NT}, \mathcal{S}, \mathcal{P})$ [7], i.e. (terminal symbols, nonterminal symbols, start symbol, productions). Figure 4 shows the grammar for workflow synthesis domain.

The grammar graph is the graph representation of the CFG. It is a directed graph transformed from DSL grammar by HISyn. It defines the search space for the code generation task, and the code generation problem is transformed to the problem of finding a subgraph called *code generation tree* (defined later) from the grammar graph.

2) Dependency graph: The dependency graph is the output of HISyn's front end. It is a directed acyclic graph. The nodes of the dependency graph are the tokens in a NL query. One token contains the word, lemma, part-of-speech (POS) tag, and named entity label. The edges of the dependency graph are the dependency relations among the tokens. The dependency graph is used as the basis of the intermediate representation in guiding code generation in the back end. Figure 3(b) shows a dependency graph of query 3 in Table I.

3) Code generation tree: A code generation tree (CGT) is a subgraph of a grammar graph. A CGT is a directed acyclic graph whose undirected counterpart is a tree. The root of a CGT is a non-terminal node that corresponds to the start symbol of the grammar. During the synthesis process, HISyn searches for the paths on the grammar graph that represent the semantic of dependency edges. The found paths are combined into CGTs, which then get transformed into code expressions as synthesis results. Figure 3(c) shows a code generation tree that represents the workflow expression in Table I(3).

B. Workflow DSL

As discussed in previous sections, HISyn [32] is the optimal synthesizer for workflow synthesis in HPC-FAIR due to its NL input, free of training and cross-domain extensibility features. As HISyn synthesizes the target DSL code expressions based on the DSL grammar and API descriptions, to synthesize the workflow, the first challenge is to define the workflow DSL. The Document generator module is designed to generate the grammar and API description files to address this challenge. This subsection first describes the APIs and variables that are used inside the DSL, and then introduces the overall DSL.

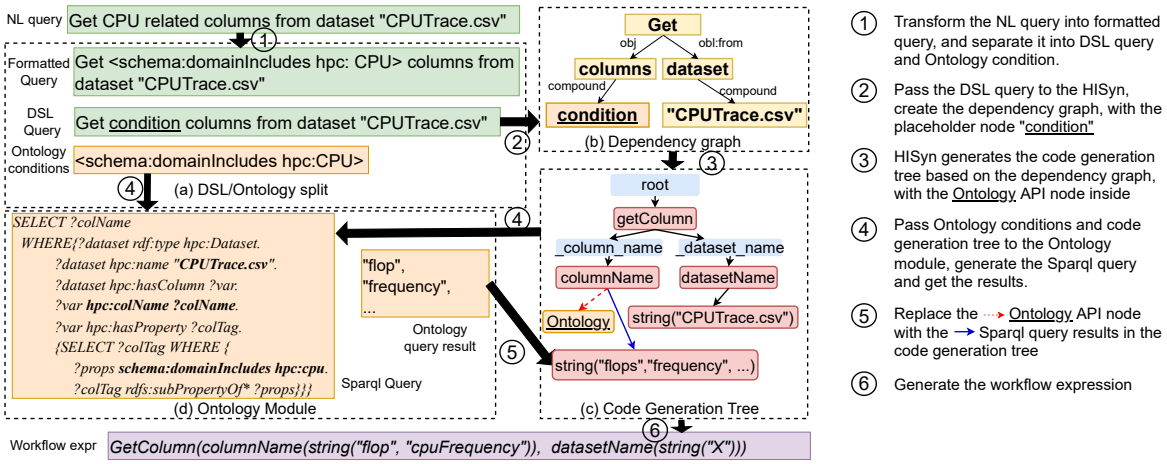

Figure 3. Running example

1) APIs and variables: APIs and variables are elements in the DSL. To generate the DSL, we need to define the APIs and variables first.

An *Ontology API*, `ontology`, is created to be mapped with the Ontology related contents inside the NL query. During the synthesis process, the Ontology contents are wrapped and mapped to this Ontology API, which serves as a placeholder inside the CGTs (Figure 3(c)). After getting the SPARQL query results from the Ontology module, the Ontology API is replaced with the results inside the CGT. The return type of this API is "_ontology". Any APIs that are related with classes or properties inside Ontology could call this API. For example, the API `columnName` takes the names of columns inside the dataset. It can either directly take a string with actual column names as input, or take the Ontology API as input since the SPARQL query returns the name of the columns.

Besides this special API, the rest of the APIs in the DSL have two categories, the data manipulation APIs and the project APIs.

Data manipulation APIs are a set of APIs defined to handle the data processing, such as fetching the data, transferring data file types and extracting specific columns from a dataset. These APIs are designed to support basic data manipulation operations that might be used by users, and fill the gap between the models and data from different projects. For example, a model M_A from project A requires a csv file as input, while the dataset D_B from project B is uploaded in a json format. If a user wants to train a model M_A with data D_B, the file type transfer API `json2csv` is necessary inside the workflow to perform the transformation. The final workflow representation is `M_A(json2csv(D_B.json))`. In Figure 4, `MergeDataset`, `GetColumn`, `json2csv` are all the data manipulation APIs.

Besides, there are also some APIs that represent the semantic of the arguments in a data manipulation API. For example, the API `columnName` is derived from the first argument (nonterminal) of API `GetColumn`, which means the first argument is the names of the dataset. These APIs are semantic APIs. They can accept either string or Ontology API as the

```
_workflow := _string | _list | _array | _file |_number | ...
_string := string(STRING) | GetHardware(_hwName) | ...
_array := GetColumn(_columnName,_datasetName)
          | MergeDataset(_csvFile, _csvFile) | …
_csvFile := json2csv(_jsonFile) | csvFile(_string)
          | logsToDataset_XPlacer(_nsight_log) |...
_datasetName := datasetName(_ontology_arg)
_columnName := columnName(_ontology_arg)
_hwName := hardwareName(_ontology_arg)
_ontology_arg:= _string | _ontology
_ontology := ontology()
...
```

API: MergeDataset *description*: Merge data in two csv files	*API*: datasetName *description*: matches dataset names *ontology_class*: hpc:dataset *ontology_subject*: hpc:name

Figure 4. Workflow DSL grammar

argument. In the CGT, if a semantic API is the parent node of the Ontology API, it will provide the query subject and scope for SPARQL query generation (see section V-B). Thus, for each semantic API, besides NL description, there are two extra entries, ontology_class and ontology_subject, to store the information needed by the Ontology module.

Project APIs are a set of APIs automatically created from the user provided data flow, such as dataset: extraction, preprocessing, and model: training, fine-tuning, inference). In HPC-FAIR, users can upload scripts used in their project's dataflow. These scripts can be used to train the model with other dataset. To synthesis workflow with these scripts, project APIs are created with *script_project* as the API name to represent these scripts in the DSL. The input, output and the description of the project APIs are extracted from the HPC Ontology. For example, in Figure 4, the XPlacer project has one script named "*logs_to_dataset*" with *nsight log file* as input and returns a *csv* file. The project API `logsToDataset_XPlacer` is created and added into the API documents with its description, and linked to the DSL grammar based on its input and output. The document generator can automate the project API creation, directly create and update the project APIs based on updates in the HPC Ontology.

2) Overall DSL: The context free grammar (CFG) is used to represent the DSL. The partial of the DSL is shown in Figure 4. The terminals $\mathcal{T}$ of the DSL are all APIs described above. The start symbol is the nonterminal $nt_{workflow}$ that

represents a workflow expression. Then the next step is to decide the nonterminals $\mathcal{NT}$ and their production rules $\mathcal{P}$.

The input and output information is used as the nonterminals to connect the APIs. Specifically, for an API A_i, its return type becomes a nonterminal nt_{return}, and the A_i itself becomes one of the production rules of nt_{return}. Then the input information of A_i also becomes a nonterminal nt_{input} and becomes the parameter of the A_i. In all, the basic syntax of a production rule is $nt_{return} := A_i(nt^*_{input})$, where * means an API can have multiple inputs. All the APIs that return the same type are derived from the same nonterminal.

For example, in Figure 4 the data manipulation API `json2csv` takes a json file as input, and returns a csv file. The project API `logsToDataset_XPlacer` takes an nsight log file as input, and also returns a csv file. Therefore, the $nt_{\_csvFile}$ can derive both APIs, and each API has their own nonterminal ($nt_{\_jsonFile}$ and $nt_{\_nsight\_log}$) as the input.

Besides the DSL grammar, the API documentation is also needed for HISyn to synthesize the workflow expression. We write NL descriptions for each data manipulation API, and extract the script descriptions from the HPC ontology as the description of the project APIs.

Please note that all the knowledge from the Ontology that feeds to INPOWS is supposed to be created when users upload the datasets and models to HPC-FAIR to enhance the interoperability and reusability. INPOWS does not require extra efforts from data providers.

C. Workflow synthesis

With grammar and API documentation settled, INPOWS then starts synthesizing the workflow expression with HISyn. Figure 3 shows the workflow synthesis process of the workflow query in Table I(3). The following steps refer to the step numbers shown in Figure 3.

Step 1. DSL/Ontology query split: As previously mentioned, another challenge is that a NL workflow query could involve two search spaces, the DSL and the HPC-FAIR Ontology. Thus, when receiving the NL query, the first step is to parse the query and split the DSL and Ontology contents. The DSL/Ontology query split module (section V-A) finds the possible ontology related components by analyzing the NLP parsing results and compare it with descriptions of Ontology concepts and properties. The matched concepts or properties are then replaced with special formats. The final queries that are annotated with Ontology components are called **formatted queries**. All the formatted queries are shown to the user and let them select the correct query through an interactive process.

Then the split module extracts the Ontology component inside the formatted query, and replaces it with a placeholder word, which is mapped to the *Ontology API* directly in the later synthesis steps. A query with placeholder is called a **DSL query**, since it only contains the information related to DSL. The Ontology components are passed to the Ontology module to generate the SPARQL queries that acquire knowledge from Ontology. Now the key elements in the query that are related to DSL and Ontology are split. They are processed separately in the following steps. Figure 3(a) shows the query parsing process. The detailed description of this module will be given in section V-A.

```
1  import os
2  import logFile
3  def logsToDataset_XPlacer(_nsight_log):
4      command = f'python3 xx/XPlacer/script/log_to_dataset.py {_nsight_log}'
5      os.system(command)
6
7  if __name__ == '__main__':
8      logsToDataset_XPlacer(logFile('nsight.log'))
```

Figure 5. Workflow implementation in Python

Step 2 and 3. HISyn Synthesis: The DSL query sentence is passed to the HISyn. HISyn generates the dependency graph (Step 2, Figure 3(b)) that shows the relation of the key information inside the DSL query, and synthesize the code generation tree (CGT) (Step 3) in the DSL with the *Ontology API* (Figure 3(c)).

Step 4. Acquire knowledge from Ontology: The knowledge is acquired from Ontology through the Ontology module (Figure 3(d)). The inputs of Ontology module are Ontology conditions from DSL/Ontology split and the CGT from HISyn. The module automatically checks the information inside the inputs, creates the SPARQL query that searches for the knowledge inside the Ontology, and returns the ontology query results. The detailed description will be given in section V-B.

Step 5 and 6. Workflow expression generation: The SPARQL query results are used to replace the Ontology API in the CGT. With the CGT that contains all the information, HISyn then generates the final workflow expression.

D. Workflow implementation and execution

Next, INPOWS executes the workflow to fulfill users' intention. The data manipulation APIs are defined and implemented by ourselves. On the contrary, the project APIs needs to be implemented to run these scripts.

The implementation of project APIs are also automatically generated by the document generator when defining these APIs from Ontology. While uploading scripts to HPC Ontology, besides the name, inputs and outputs, the script path, file type and the running command for the scripts should also be provided as suggested by HPC-FAIR. For example, a python script *logs_to_dataset.py* is a data processing step in project *XPlacer*. This path of the script is *xx/XPlacer/script/log_to_dataset.py*, and its execution command is "*python3 log_to_dataset.py param1*". Then the implementation of API `logsToDataset_XPlacer` (INPOWS uses python for workflow implementation) is shown in Figure 5 line 3-5. A bash command is created by joining the execution command, path to script and arguments, then executed with system package. While executing, it only needs to import the APIs involved in the workflow expression, and run the workflow expression in a python script, such as line 7-8 in Figure 5.

V. Interaction with Ontology

The previous section IV-C introduces the overall workflow synthesis process. It briefly mentioned two modules

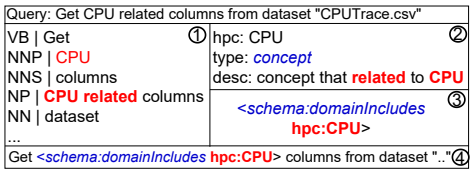

Figure 6. DSL/Ontology query split

that interact with Ontology: the DSL/Ontology module that identifies and formats the Ontology related components inside the original NL query; the Ontology Module that creates the SPARQL queries and acquires knowledge inside Ontology. This section describes these two modules in more detail.

A. DSL/Ontology query split

INPOWS uses natural language (NL) as the input, which provides flexibility and convenience to the user. However, it also brings complexity to the synthesis task. Compared to traditional program synthesis whose search space is a target DSL, INPOWS faces an additional search space, Ontology. Thus, it is essential to differentiate the components inside the NL query that refer to the DSL and Ontology, split and send them to designated modules for further processing. The DSL/Ontology split module (i.e. the split) is designed to deal with such tasks.

Currently INPOWS supports two types of ontology related workflow queries, the **property query** and the **concept query**. The property query searches for the subject with certain properties. These properties usually link to the corresponding Ontology properties such as *hpc:license* and *hpc:subject*. For example, in Table I(2), it queries the property *hpc:subject*, and the replacement of the ontology component in the query is *<hpc:subject "GPGPU">*.

The concept query searches for the hierarchical information from the Ontology concepts. These concepts usually link to certain columns of datasets as related property domains of the columns. For example, in Figure 1, the RDF triplet *(column_1, hpc:hasProperty, hpc:cpuFrequency)* indicates that *column_1* is linked to a property `hpc:cpuFrequency`. Moreover, "hpc:cpuFrequency" is a property used to describe a concept named "hpc:CPU" as indicated by the triplet of *(hpc:cpuFrequency, schema:domainIncludes, hpc:CPU)*. Therefore, if the user wants to search for columns that are related with the concept of CPU, we can get *column_1* and its name *"frequency"* from Ontology.

To support these two types of queries, the query split firstly needs to identify the Ontology components in the query. Then it replaces the Ontology components with special tokens, so that the split Ontology and DSL portions in the query can be processed by designated modules of INPOWS . The following subsections explains the details of these two steps.

1) Identify and map Ontology components: After receiving the NL queries, the first step is to identify the Ontology components. The split first parses the query with an NLP parser. The parser labels the part-of-speech (POS) for the query based on the English grammar. Based on our observation in HPC Ontology, the Ontology properties and concepts are typically *nouns*, *noun phrases* and *verbs*. Thus the split extracts the words and phrases with these POS annotations. These words and phrases become the **Ontology candidates**. For example, Figure 6(1) shows the potential words and phrases that might refer to knowledge in Ontology.

Both Ontology properties and concepts are called **Ontology tags.** Now the goal is to find the correct Ontology tags for each Ontology candidate. Each Ontology tag has a NL description to describe its semantics, as shown in Figure 6(2). The split maps the candidates to tags based on these NL descriptions. Specifically, in each Ontology candidate, if a word or its synonyms are also inside the description of an Ontology tag, this tag is a mapping of that candidate words. The word mapping is case-insensitive, and it compares the words with lemmatization and stemming to improve the robustness of mapping. For example, in Figure 6(1), the noun word *CPU* is mapped to Ontology concept *hpc:CPU*. Besides, inside an Ontology candidate phrase, if more than one word can be mapped to the same Ontology tag, then this tag is more likely to represent the meaning of the matched portion of the phrase. For example, in Figure 6(1), the first two words in the noun phrase *CPU related columns* are mapped to tag *hpc:CPU*.

2) Generate formatted query: With all possible mappings found, INPOWS then replaces the Ontology candidates in the original NL query with the corresponding Ontology tags. In the NL query parse tree, a phrase is the parent node of the words inside it. If the parent node and the child nodes mapped to the same Ontology tag, INPOWS replaces inside the parent (phrase) node. For example, in Figure 6, INPOWS replaces the mapped *CPU related* inside the noun phrase instead of the noun word *CPU* only.

If the mapped tag is an Ontology concept, it queries the columns in the datasets that are related to this concept. Such concept and sub-concept relations are defined with the property *schema:domainIncludes* inside HPC Ontology. Thus, this property is added in the front of the Ontology concept tag to find matches for all sub-concepts and concepts. (e.g. *<schema:domainIncludes hpc:CPU>* in Figure 6(3)).

If the mapped tag is an Ontology property, the words that meet both the following two conditions are concatenated to the tag as the value of this property: (1) words that follow the Ontology candidate in the query, and (2) words that are siblings of the Ontology candidate in the parse tree. For example, in Table I(2), the "GPGPU" is the sibling of the "subject" in the parse tree. With the "subject" mapped to *hpc:subject* tag, the final Ontology tag after concatenation is *<hpc:subject "GPGPU">*.

An NL query can have multiple Ontology candidates that are mapped to Ontology tags. Some candidates could map to multiple tags and some are not supposed to map to any tags. Thus, INPOWS creates all the possible combinations for the mapped Ontology tags, as each combination includes one Ontology tag from one candidate. For each combination, it replaces the Ontology candidates with the Ontology tags in the NL query, wrapped with a pair of angle brackets "<>". The new queries with *"<Ontology tags>"* are called **formatted**

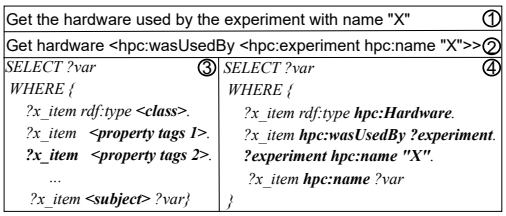

Figure 7. SPARQL template for property queries

query (e.g. Figure 6(4)).

In the end, there will be a list of formatted queries. The formatted query that has tags from all the Ontology candidate tags is the **default formatted query**. The first generated query becomes the default formatted query if there are several formatted queries that meet this condition. INPOWS shows the list to the user, and let them select the one that can represent their intents. Please note that the formatted query list always includes the original NL query without any tags.

With the correct formatted query selected, INPOWS then splits the query into two parts. The *"<Ontology tags>"* inside the formatted query is replaced with the placeholder word "*condition*". The new query has no content related to Ontology, hence its search space limited to the DSL. We call it a **DSL query**. The DSL query is directly passed to the front end of HISyn, while the placeholder word "*condition*" is directly mapped to the *Ontology API* in the synthesis process.

All the *<"Ontology tags">* inside the formatted query are then passed to the Ontology Module to create the SPARQL query that acquires the information inside the Ontology.

B. Ontology Module

The Ontology module plays a key role in communication with Ontology. It takes the Ontology tags from the formatted NL query and the CGT from HISyn synthesis process as input, automatically generates the SPARQL query that can acquire the knowledge from Ontology. After getting the query results, it replaces the *Ontology API* inside the CGT, return the final CGT to the back end of HISyn, and generate the final workflow expression. This section gives a detailed description of the Ontology module.

1) Generate SPARQL query: As mentioned in section V-A, for Ontology queries, INPOWS currently supports property queries and concept queries. Therefore, two SPARQL templates are prepared to support these two query types. The template for property queries are shown in Figure 7(3). The filled template for concept query is shown in Figure 3(d), and the highlighted components are the Ontology tags and information from the CGT.

The types of queries are identified through Ontology tags from the formatted NL query. If a tag has *schema:domainIncludes*, it searches for the related concept of dataset columns, hence it is the concept query. Otherwise, it queries for certain properties, hence it is the property query. With the query type determined, INPOWS then fills the template with the *<"Ontology tags">* and information from CGT.

For concept query template, from the top to bottom, there are three blanks that need to be filled, *dataset name, query subject* and *Ontology concept tag*. The dataset name is from the CGT by checking the child node of API `datasetName`, while the query subject is determined by the parent of the *Ontology API*. The *Ontology concept tags* are from the DSL/Ontology split step directly. For example, in Figure 3(d), the *dataset name* and *query subject* are filled by the information from CGT. Specifically, `datasetName` indicates the *dataset name*. The Ontology API is the child of `columnName`, thus it indicates the *query subject*. The *Ontology concept tag* is from the formatted DSL query directly.

For property query template, the blanks are *class*, *query subject* and *Ontology property tags*. The *class* and the query subject is filled by the parent node of the Ontology API, while the properties are filled by the *Ontology property tags* from the formatted query. For example, the parent node of the Ontology API is `hardwareName` in the CGT of the query in Figure 7(2). Thus, the *class* of the template is *hpc:Hardware*, and the *query subject* is *hpc:name*. Next INPOWS fills the property tags inside the template. The Ontology property tag has two embedded tags. Inside the inner tag, the *hpc:experiment* is a Ontology class mapped by the NL query keyword experiment. Since the goal is to search for instances with certain properties, the Ontology class is replaced with a SPARQL variable *?experiment*. Then the inner tag (*?experiment hpc:name "X"*) is a valid triplet. They are put into the template from left to right. The object of the outer tag is the subject of the inner tag, i.e. *hpc:wasUsedBy ?experiment*. Then the inner tag is directly filled into the template as *property tags 2* inside the template. At this step, the full SPARQL query is completed as shown in Figure 7(4).

2) Replace Ontology API: After generating the SPARQL query, INPOWS then runs it to acquire the knowledge from Ontology. The concept query searches for not only the given concept, but all the sub-concepts of the given concept, and return the columns that relate to all the concept and sub-concepts. In Figure 3(d), the column "flops" and "frequency" are two columns that are related to the sub-concepts of "CPU".

Then query results are used to replace the Ontology API node inside the CGT. In Figure 3(5), it replaces the Ontology API with actual column names "flops" and "frequency". Then HISyn generates the corresponding workflow expression from the CGT. In Table I(2), the dataset name "lassen_overhead" and "performance_result_dataset" are also the SPARQL query results.

VI. Experiments

We conduct a set of experiments to examine the efficacy of INPOWS to synthesis workflows from HPC-FAIR. We use the experiments to answer the following four research questions: (1) What is the accuracy of the DSL/Ontology splitting module? (2) What is the overall accuracy of the workflow synthesis framework?(3) What is the accuracy of creating the SPARQL query? (4) What are the reasons that cause errors?

We describe the experiment settings in Section VI-A, report our experiment results in Sections VI-B, and provide a detailed error analysis in several representative cases in Section VI-C.

A. Methodology

1) Dataset: We collect 60 NL queries to evaluate the INPOWS. The queries can be divided into three categories based on their query type.

Data manipulation queries are queries that only need to process the data. It does not require interaction with Ontology. There are 17 data manipulation queries in total.

Ontology-interactive queries are queries that mainly search for the information inside Ontology. There are 25 Ontology queries.

Combined queries are combinations of the previous queries. Inside one NL query, it requires both data manipulation and Ontology query. There are 18 combined queries.

Table I shows three example queries from each category and their corresponding workflow expression.

All the queries are provided by a group of contributors of the HPC-FAIR [55]. We provided the API documentation to show the scope of the supported operations. Besides, Table I also provide one example query for each category as the reference.

We use HPC Ontology from [23] in our experiments.

2) Evaluation Metrics: We use all the test cases in each category in the experiments. We use the DSL/Ontology split accuracy to evaluate the DSL/Ontology split module. The DSL/Ontology split accuracy is the ratio between the number of the correct formatted query and the number of total test cases. The formatted query is the query selected by the users.

We use workflow expression synthesis accuracy to evaluate the performance of INPOWS. The expressions are synthesized from NL queries. The synthesis accuracy is the ratio between the number of *correctly* synthesized DSL code expressions and the number of total test cases. A synthesized DSL code is *correct* if it performs the intent expressed inside the NL query.

3) Methods for comparison: We use the DSL/Ontology split accuracy before the user interaction as the comparison to evaluate the split module. This accuracy is the ratio between the number of the correct default formatted query (V-A) and the number of total test cases. The formatted queries are the queries selected by the users.

Since INPOWS is the first workflow synthesizer that allows NL queries, it is hard to compare it with existing works. We compare the workflow synthesis accuracy with *correctly formatted queries synthesis accuracy* for all test cases. It is the accuracy of synthesis by providing the correct formatted query to HISyn and Ontology modules. We use the *w/o-Ontology support synthesis accuracy* as the baseline, which is the synthesis accuracy when removing the Ontology module.

B. Results

(Q1) The experiment is shown in Figure 8. Figure 8(a) shows the DSL/Ontology Split accuracy . For all test cases, with the help of users, the split achieves 95% accuracy on providing correct formatted queries. In comparison, without interaction, only 65% of default formatted queries are correct. Within 43 queries that involve Ontology (Ontology interactive queries and combined queries), the split correctly generates 40 formatted queries with the help of the user interaction, achieving 93.02% accuracy. In comparison, without interaction, only 32 default formatted queries are correct, achieving 74.42% accuracy. The difference between with and without user interaction (18.6% for Ontology queries and 35% for all queries) shows large accuracy improvements brought by the user interactions, demonstrating the effectiveness of the user interaction to resolve the NL ambiguity.

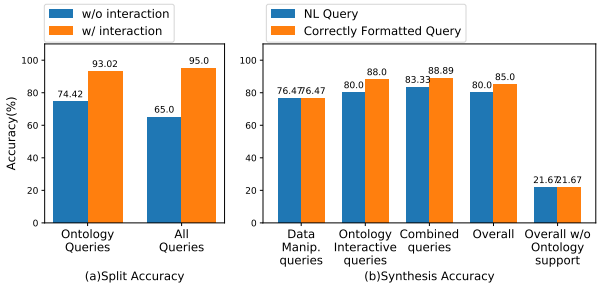

Figure 8. DSL/Ontology split accuracy and synthesis accuracy

The accuracy of "All queries" is lower than "Ontology queries" without interaction, and is higher with interaction. It is because there are 17 data manipulation queries that do not involve Ontology. During mapping, 10 of them have one or more words mapped to Ontology tags, which lower the overall accuracy. With interaction, users select the un-tagged queries inside the query list, which results in a higher accuracy.

(Q2) Figure 8(b) shows the overall synthesis accuracy. If the Ontology module is not involved, every query that needs knowledge from Ontology is become incorrect. Thus it only achieves 21.67% accuracy. With the Ontology module gets involved, the overall synthesis accuracy becomes 80.33%. For three query categories, The data manipulation queries, Ontology interactive queries and the combined queries achieve 76.47%, 80% and 83.33% accuracy respectively. The failed cases are caused by wrong Ontology tags and wrong dependency structures.

We provide the correct formatted query to INPOWS as the comparison for synthesis accuracy. With Ontology tags correctly mapped, the overall synthesis accuracy increases 5%. For two Ontology related queries, the accuracy increases 8% and 5.56% respectively. It shows that with the query split errors fixed, INPOWS can achieve higher accuracy.

(Q3) As for the Ontology module, we are using the templates to create the SPARQL query. In our experiment, for all the ontology related queries, as the Ontology tags are correctly split and the CGT is correctly generated, the SPARQL query is also successfully created and can provide correct knowledge from Ontology.

C. Error analysis

(Q4) In this section, we conduct the case study to analyze the reason for failed cases. We first analyzed 1 reason where the DSL/Ontology split module failed to give correct Ontology

tags. Then we analyze another 2 reasons that cause errors in the synthesis process.

1) DSL/Ontology split errors: There are 1 reasons that lead to incorrect formatted query.

Reason-1: Wrong POS annotations

Query-1: Extract data linked to *cpu (POS: JJ)* from file "X".

In the DSL/Ontology split module, we map the nouns, noun phrases and verbs to the Ontology tags. Thus, if a wrong POS annotation is given by NLP parser, the words and phrases may not be mapped to the Ontology tags. Query-1 is an example of wrong POS tags. The NLP parser annotates the word "cpu" with POS "JJ" (i.e. adjective). Then the word "cpu" is not mapped due to the error POS annotation. One way to solve this error is to provide the English sentence with precise grammar. For example, adding "the" in front of the word "cpu" directly addresses the issue.

2) Workflow synthesis error: There are 2 reasons that lead to incorrect synthesis results.

Reason-2: Unmatched information between query and workflow expression

Query-2: Get the multiplication results of columns X and Y from dataset Z.

The correct workflow expression for this query should be `DotProduct(GetColumn(..(X),dataset(Z)), GetColumn(..(Y)),dataset(Z)))`. In this expression, the `dataset(Z)` becomes the argument of both `GetColumn`. However, during the synthesis process, one key information could be mapped to an API once. Therefore, only one `dataset(Z)` exists in the synthesis result, which is incorrect. One way to address this issue is to provide the information for each *columns*, e.g. "Get the multiplication results of columns X from dataset Z and column Y from dataset Z". The other way is to improve HISyn to support this semantic, which is beyond the scope of this paper.

Reason-3: Wrong dependency graph.

Query-3: Select X1 and X2 from dataset A and Y1 and Y2 from dataset B, then merge them into dataset C.

The dependency graph is the graph representation of key information of the original NL query. If the dependency graph is incorrect, it will be hard for HISyn to synthesize the correct results. Query-3 includes three tasks inside the one workflow query, two tasks that extract columns from different datasets, then one task that merges the extracted data. However, in the dependency graph of this query, the word "select" becomes the parent node of the word "merge". Hence in the synthesized CGT, the API `GetColumns` (for "select") becomes the parent node of the API `MergeData` (for "merge"), which result in the reverse execution of the original intent. To address this issue, we could split this query into separate sub-queries. The first two queries generate data extraction workflow individually, and the results of them are merged by the workflow from the third query.

VII. Related Works

There have been attempts made in various domains such as Bio-infographics, Spectrometry, Computational biology and Pharmacogenomic to automatically generate inter-operable workflows [28], [31], [42], [56], [60]. Common workflow language (CWL) [2] is one such example, which makes data analysis workflows portable for all by standardizing the computational reuse. CWL community has developed tools, software libraries, specifications, and has shared CWL descriptions for popular tools [14], [35], [52], [54].However, the use of CWL usually requires that users know the syntax of the language as well as fully understand the standardized vocabulary to automate workflows. In comparison, the NL input of INPOWS provides more flexibility to users.

Ontology have been used in this area for the development of various projects like loose programming [17], workflow generation [48], [49], [59] and meta analysis of data-mining tasks [15], [16], [39]. They provide common vocabulary to store the domain knowledge. INPOWS too makes use of the HPC-ontology to retrieve essential information about the HPC domain. Several researches have proposed translation of NL into SPARQL [8], [37], [53], [63]. AutoSPARQL [22], which uses supervised machine learning to generate SPARQL queries, presents one such example. However, the SPARQL query is not sufficient for tasks in HPC-FAIR. Besides querying knowledge from Ontology, INPOWS takes advantage of the APIs in workflow DSL which could directly access the data files and model script.

Another body of work is Natural Language (NL) based program synthesis. [32]–[34] synthesis target code expression using the NLU-driven approach. Many recent studies have pursued modern machine learning for NL programming [3], [6], [9]–[13], [18], [24]–[26], [40], [43], [44], [47], [51], [62]. However, applying these approaches to program analysis would require many training examples to cover the vast space of possible code complexities and situations. They are difficult to apply to areas where labeled training data is scarce, while HPC-FAIR is such an example.

VIII. Conclusion

This paper proposed INPOWS, the first workflow synthesizer that is Ontology-based and allows NL queries. INPOWS seamlessly integrates Ontology with NLU-driven workflow synthesis, empowers flexible NL input for users and superior extensibility to adopt domains with continuous changes. The interactive design further resolve the NL ambiguities. The paper applied INPOWS to HPC-FAIR, a FAIR practice for HPC's ML models and datasets. It demonstrates the effectiveness of the INPOWS through a set of experiments, showing 80% synthesis accuracy for 60 NL queries. The interactive design achieves 95% accuracy for extracting Ontology information inside the NL query, demonstrating the robustness of the INPOWS .

Acknowledgment

Prepared by LLNL under Contract DE-AC52-07NA27344 and supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (LLNL-CONF-833524).

REFERENCES

[1] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *IEEE International Conference onServices Computing, 2004.(SCC 2004). Proceedings. 2004*, pages 23–30. IEEE, 2004.

[2] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common workflow language, v1. 0. figshare, 2016.

[3] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. Autopandas: neural-backed generators for program synthesis. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[4] Liming Chen, Nigel R Shadbolt, Carole Goble, Feng Tao, Simon J Cox, Colin Puleston, and Paul R Smart. Towards a knowledge-based approach to semantic service composition. In *International Semantic Web Conference*, pages 319–334. Springer, 2003.

[5] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 487–502, 2020.

[6] Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 602–612, 2019.

[7] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[8] Danica Damljanović, Valentin Tablan, and Kalina Bontcheva. A text-based query interface to owl ontologies. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, 2008.

[9] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356. ACM, 2016.

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[11] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[12] Tihomir Gvero and Viktor Kuncak. Synthesizing java expressions from free-form queries. In *Acm Sigplan Notices*, volume 50, pages 416–432. ACM, 2015.

[13] Gang Hu, Linjie Zhu, and Junfeng Yang. Appflow: using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 269–282, 2018.

[14] Gaurav Kaushik, Sinisa Ivkovic, Janko Simonovic, Nebojsa Tijanic, Brandi Davis-Dusenbery, and Deniz Kural. Rabix: an open-source workflow executor supporting recomputability and interoperability of workflow descriptions. In *PACIFIC SYMPOSIUM ON BIOCOMPUTING 2017*, pages 154–165. World Scientific, 2017.

[15] C Maria Keet, Agnieszka Lawrynowicz, Claudia d'Amato, and Melanie Hilario. Modeling issues & choices in the data mining optimization ontology. 2013.

[16] C Maria Keet, Agnieszka Ławrynowicz, Claudia d'Amato, Alexandros Kalousis, Phong Nguyen, Raul Palma, Robert Stevens, and Melanie Hilario. The data mining optimization ontology. *Journal of web semantics*, 32:43–53, 2015.

[17] Johannes F Kruiger, Vedran Kasalica, Rogier Meerlo, Anna-Lena Lamprecht, Enkhbold Nyamsuren, and Simon Scheider. Loose programming of gis workflows with geo-analytical concepts. *Transactions in GIS*, 25(1):424–449, 2021.

[18] Gregory Kuhlmann, Peter Stone, Raymond Mooney, and Jude Shavlik. Guiding a reinforcement learner with natural language advice: Initial results in robocup soccer. In *The AAAI-2004 workshop on supervisory control of learning and adaptive systems*. San Jose, CA, 2004.

[19] Anna-Lena Lamprecht. User-level workflow design. *Lecture Notes in Computer Science*, 8311, 2013.

[20] Anna-Lena Lamprecht, Tiziana Margaria, and Bernhard Steffen. Bio-jeti: a framework for semantics-based service composition. *BMC bioinformatics*, 10(10):1–19, 2009.

[21] Anna-Lena Lamprecht, Stefan Naujokat, Bernhard Steffen, and Tiziana Margaria. Constraint-guided workflow composition based on the edam ontology. *Nature Precedings*, pages 1–1, 2010.

[22] Jens Lehmann and Lorenz Bühmann. Autosparql: Let users query your knowledge base. In *Extended semantic web conference*, pages 63–79. Springer, 2011.

[23] Chunhua Liao, Pei-Hung Lin, Gaurav Verma, Tristan Vanderbruggen, Murali Emani, Zifan Nan, and Xipeng Shen. Hpc ontology: Towards a unified ontology for managing training datasets and ai models for high-performance computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 69–80. IEEE, 2021.

[24] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[25] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*, 2018.

[26] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[27] Phillip Lord, Pinar Alper, Chris Wroe, Robert Stevens, Carole Goble, Jun Zhao, Duncan Hull, and Mark Greenwood. The semantic web: Service discovery and provenance in my-grid. In *W3C Workshop on Semantic Web for Life Sciences*. Citeseer, 2004.

[28] Anthony Mammoliti, Petr Smirnov, Zhaleh Safikhani, Wail Ba-Alawi, and Benjamin Haibe-Kains. Creating reproducible pharmacogenomic analysis pipelines. *Scientific Data*, 6(1):1–7, 2019.

[29] Tiziana Margaria and Bernhard Steffen. Backtracking-free design planning by automatic synthesis in metaframe. In *International Conference on Fundamental Approaches to Software Engineering*, pages 188–204. Springer, 1998.

[30] Emanuela Merelli, Giuliano Armano, Nicola Cannata, Flavio Corradini, Mark d'Inverno, Andreas Doms, Phillip Lord, Andrew Martin, Luciano Milanesi, Steffen Möller, et al. Agents in bioinformatics, computational and systems biology. *Briefings in bioinformatics*, 8(1):45–59, 2007.

[31] Steffen Möller, Stuart W Prescott, Lars Wirzenius, Petter Reinholdtsen, Brad Chapman, Pjotr Prins, Stian Soiland-Reyes, Fabian Klötzl, Andrea Bagnacani, Matúš Kalaš, et al. Robust cross-platform workflows: how technical and scientific communities collaborate to develop, test and share best practices for data analysis. *Data Science and Engineering*, 2(3):232–244, 2017.

[32] Zifan Nan, Hui Guan, and Xipeng Shen. Hisyn: Human learning-inspired natural language programming. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foun- dations of Software Engineering (ESEC/FSE)*, 2020.

[33] Zifan Nan, Hui Guan, Xipeng Shen, and Chunhua Liao. Deep nlp-based co-evolvement for synthesizing code analysis from natural language. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*, pages 141–152, 2021.

[34] Zifan Nan, Xipeng Shen, and Hui Guan. Enabling near real-time nlu-driven natural language programming through dynamic grammar graph-based translation. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 278–289. IEEE, 2022.

[35] Rupert W Nash, Michael R Crusoe, Max Kontak, and Nick Brown. Supercomputing with mpi meets the common workflow language standards: an experience report. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, pages 17–24. IEEE, 2020.

[36] Stefan Naujokat, Anna-Lena Lamprecht, and Bernhard Steffen. Loose programming with prophets. In *International Conference on Fundamental Approaches to Software Engineering*, pages 94–98. Springer, 2012.

[37] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. Natural language processing and program analysis for supporting todo comments as software evolves. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[38] Magnus Palmblad, Anna-Lena Lamprecht, Jon Ison, and Veit Schwämmle. Automated workflow composition in mass spectrometry-based proteomics. *Bioinformatics*, 35(4):656–664, 2019.

[39] Pance Panov, Sašo Džeroski, and Larisa Soldatova. Ontodm: An ontology of data mining. In *2008 IEEE International Conference on Data Mining Workshops*, pages 752–760. IEEE, 2008.

[40] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[41] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *International semantic web conference*, pages 30–43. Springer, 2006.

[42] Stephen R Piccolo, Zachary E Ence, Elizabeth C Anderson, Jeffrey T Chang, and Andrea H Bild. Simplifying the development of portable, scalable, and reproducible workflows. *Elife*, 10:e71069, 2021.

[43] Illia Polosukhin and Alexander Skidanov. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

[44] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, 2015.

[45] MD R-Moreno and Paul Kearney. Integrating ai planning techniques with workflow management system. *Knowl-Based Syst*, 15(5-6):285–291, 2002.

[46] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *International Workshop on Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2004.

[47] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *IJCAI*, pages 792–800, 2015.

[48] Leonardo Salayandia, Steve Roach, and Ann Q Gates. Program synthesis from workflow-driven ontologies. In *NAFIPS 2008-2008 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 1–6. IEEE, 2008.

[49] Mumtaz Siddiqui, Alex Villazon, and Thomas Fahringer. Semantic-based on-demand synthesis of grid activities for automatic workflow generation. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 43–50. IEEE, 2007.

[50] Bernhard Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, 1993.

[51] Yu Su, Ahmed Hassan Awadallah, Madian Khabsa, Patrick Pantel, Michael Gamon, and Mark Encarnacion. Building natural language interfaces to web apis. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 177–186. ACM, 2017.

[52] Denis Torre, Alexander Lachmann, and Avi Ma'ayan. Biojupies: automated generation of interactive notebooks for rna-seq data analysis in the cloud. *Cell systems*, 7(5):556–561, 2018.

[53] Thanh Tran, Philipp Cimiano, Sebastian Rudolph, and Rudi Studer. Ontology-based interpretation of keywords for semantic search. In *The semantic web*, pages 523–536. Springer, 2007.

[54] Wil MP Van Der Aalst and Arthur HM Ter Hofstede. Yawl: yet another workflow language. *Information systems*, 30(4):245–275, 2005.

[55] Gaurav Verma, Murali Emani, Chunhua Liao, Pei-Hung Lin, Tristan Vanderbruggen, Xipeng Shen, and Barbara Chapman. Hpcfair: Enabling fair ai for hpc applications. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 58–68. IEEE, 2021.

[56] Liya Wang, Zhenyuan Lu, Peter Van Buren, and Doreen Ware. Sci-apps: a cloud-based platform for reproducible bioinformatics workflows. *Bioinformatics*, 34(22):3917–3920, 2018.

[57] Mark Wilkinson, Benjamin Vandervalk, and Luke McCarthy. The semantic automated discovery and integration (sadi) web service design-pattern, api and reference implementation. *Nature Precedings*, pages 1–1, 2011.

[58] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

[59] David Withers, Edward Kawas, Luke McCarthy, Benjamin Vandervalk, and Mark Wilkinson. Semantically-guided workflow construction in taverna: the sadi and biomoby plug-ins. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 301–312. Springer, 2010.

[60] Haoqi Zhang, Eric Horvitz, and David C Parkes. Automated workflow synthesis. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[61] Jie Zhang, Jining Yan, Yan Ma, Dong Xu, Pengfei Li, and Wei Jie. Infrastructures and services for remote sensing data production management across multiple satellite data centers. *Cluster Computing*, 19(3):1243–1260, 2016.

[62] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.

[63] Qi Zhou, Chong Wang, Miao Xiong, Haofen Wang, and Yong Yu. Spark: Adapting keyword query to semantic search. In *The semantic web*, pages 694–707. Springer, 2007.