International Conference on Applied Sciences IOP Publishing
Journal of Physics: Conference Series **1426** (2020) 012033 doi:10.1088/1742-6596/1426/1/012033

Principles of development of product lifecycle management system for threaded connections based on the Python programming language

V B Kopei, O R Onysko and V G Panchuk

Ivano-Frankivsk National Technical University of Oil and Gas, Department of Computerized Mechanical Engineering, Karpatska str., no. 15, 76019 Ivano-Frankivsk, Ukraine

E-mail: vkopey@gmail.com

Abstract. The principles of the development of PLM-system for threaded connections of oil and gas equipment are described. In order to ensure efficiency, the PLM-system must be isomorphic to other complex systems and have their laws, in particular, the emergence and “requisite variety”. The proposed system belongs to the class of hybrid multi-agent intelligent systems that combine various methods of knowledge representation and decision making. The system contains a knowledge base with inference rules, an inference engine, a code editor, simulation models of threaded connections, the results of their simulations, and other components. The knowledge base contains facts, in which the factors that affect the reliability and durability of threaded connections, are connected by cause-effect relations "is the cause" and "is the effect". Facts can have such properties as an information source, dependence of quantities, a simulation model, etc. The rules of inference allow you to get new facts from the knowledge base and simulation models. All system components or their software interfaces are developed in a high-level general-purpose programming language Python, which simplifies the implementation of the system and the integration of components with different types. In particular, any Python package can be easily connected to the system. Classes and individuals of ontology are declaratively described by Python classes and objects, attributes and relationships are Python-attributes. For easy editing, the system code is divided into parts, which are combined before execution.

1. Statement of the problem

PLM-systems are intended for informational support of all phases of the product life cycle (concept, design, manufacturing and implementation) [1], [2]. Such systems should contain information components for making optimal decisions at each phase. Modern products can combine thousands of high-tech solutions that have been gradually developed and improved over the years. Due to the complexity of the PLM domain knowledge, these information systems need to be classified as complex systems [2]. In accordance with the principle of isomorphism [3], they must have all laws of complex systems (emergence, “requisite variety”, nonlinearity, spontaneous order, feedback loops, self-similarity, historicity, adaptation and others). Developers of PLM systems rarely focus on this [2]. Partially, these laws are used in systems engineering [4] and systems analysis techniques. In particular, the laws of emergence, historicity, “requisite variety” and spontaneous order are noticeable in the systems analysis method "gradual formalization of the decision-making model" [5]. The authors of the method note that most of projects of complex systems need to be described by a class of self-organizing systems, models of which must be constantly adjusted and developed. According to the laws of emergence and "requisite variety", PLM should be a hybrid intelligent system that combines various models of intelligent systems that complement each other. A large list of such models is given in [6].

To be isomorphic to complex systems, a PLM-system must have base properties: a large number of heterogeneous elements (intellectual agents), autonomy of elements, decentralization, interactions between elements, simple local rules (behavior) of an element. These requirements correspond to such software systems as MAS (multi-agent system) [7], [8] and the AOP (agent-oriented programming) approach [9] to their development. MAS can simplify the development and application of PLM [10] and also, thanks to a systems approach, increase their efficiency. For a simple description of the behavior of elements, the perspective is the integration of rule-based systems (rules engine) into the PLM [11].

To simplify system design, we suggest using a high-level, general-purpose programming language Python. It is a simple, popular, open-source language, which focused on the rapid development and system integration of heterogeneous components and has a large number of packages, in particular: pythonOCC [12] (a python wrapper for the Open Cascade Technology geometric modeling kernel), NumPy (base N-dimensional array package), SciPy [13] (fundamental library for scientific computing), Matplotlib [14] (comprehensive 2D Plotting), scikit-learn [15] (machine learning), pyDatalog [16] (logic programming), pandas [17] (data analysis), NetworkX [18] (studying complex networks). Other open-source software may be available through API: CalculiX [19] (finite-element analysis), Gmsh [20] (3D finite element mesh generator). Such integration can provide emergence and "requisite variety". Using Python can also significantly simplify the development of MAS [21]. Known general-purpose MAS for Python are osBrain [22] or PADE [23].

The aim of the work is to describe the principles of the development of PLM-system for threaded connections of oil and gas equipment, which is based on the laws of complex systems and AOP/MAS, focused on simple implementation and use, and created in Python 2.7 using the above-mentioned open-source packages.

2. Description of the system and principles of its development

The proposed PLM-system is designed to make optimal decisions in the design, manufacture, repair and operation of threaded connections. Optimal decisions can be made according to the criteria of reliability and durability of threaded connections. An example of the system is available at GitHub (https://github.com/vkopey/ThreadsPLM).

The system consists of objects (agents) whose classes inherit the `Agent` class. All objects are stored in the dictionary `KB`, where the keys are the unique names of the objects — the values of their `__name__` attributes. The tools.py module contains the `KB` dictionary and utility functions that are available for agents. Objects can be created dynamically during the user's dialogue with the system or generated by other objects. In this regard, it is more convenient to describe each object in a separate Python-module. The class name (ClassName) and the object name (AgentName) are automatically detected from the file name of this module (ClassName_AgentName.py). You can prevent name conflicts, for example, by using the `find()` function for searching names, the dictionary method `has_key`, or the `autoKey()` function for generating unique names. The `createKB`() function for each such file creates an `obj` object of the corresponding class, adds it to the `KB` dictionary using the `obj.__name__` key, and executes this file as Python code in the `obj.__dict__` namespace using `execfile()` (function for dynamic execution of Python code). This allows you to access the object attributes in this file without using the object name and thus to shorten the code.

2.1. Description of the class Agent

The base class `Agent` describes objects with a `rule()` method, an `active` attribute (determines whether the object is active) and a `__name__` attribute (object name). All other classes described below inherit the `Agent` class. The method `rule()` defines the rules of agent behavior. The method should return `True` if it has made any attribute changes. Otherwise, the method should return `False` or `None`.

In general, rules of active agents are applied as long as this leads to changes in the system. This ensures the evolution of the system and is a prerequisite for self-organization. A single call of the `rule()` method for each agent is called iteration. Iterations can be executed by the `applyRules(lst=KB, n=5)` function. If necessary, the `applyRules`() function can apply rules only to objects from list `lst`. The function parameter `n` determines the maximum number of iterations.

The `saveKB()` and `loadKB()` functions can be used to save and load agents from persistent storage. They use the `dill` package [24] and the standard module `shelve`. These functions can be useful in case of long iterations. In addition, the storage of attribute values in the source code of agents can be performed by the user.

2.2. Description of the class Factor

The `Factor` class describes agents-factors that affect the reliability and durability of threaded connections. Each factor has attributes `isCause` (is the cause of the factor), `isEffect` (is the effect of the factor), `subClassOf` (is the subclass of the factor), `isContraryOf` (is the contrary of the factor). These attributes may contain a set of factors names. Thus factors can be linked by relations using these attributes. In particular, the `isCause` and `isEffect` attributes can create cause-effect relations between factors. An example of the description of the `"corrosion damage"` factor in the "Factor_corrosion damage.py" file:

```
#-*- coding: utf-8 -*-
"""Factor_corrosion damage"""
isCause={"stress concentration"}
```

where the first line defines the character encoding of the file, the second is the object documentation (attribute `__doc__`), in the third, the `isCause` attribute is assigned a value, which indicates that the factor is the cause of the `"stress concentration"` factor.

2.3. Description of the class Property

The `Property` class describes agents-properties that are used to create relations between ontology objects. Properties have attributes that are similar to the properties of the Web Ontology Language (OWL): `inverseOf` – inverse property name, `domain` – a set of classes that have this property, `range` – a set of classes, which objects can be included in a set of property values, `FunctionalProperty` – the property has only one value if `FunctionalProperty=True` and others. For example, the `isCause` attribute of the `Factor` class is a property if the `"isCause"` agent exists and described in the "Property_isCause.py" module:

```
"""Property_isCause"""
inverseOf="isEffect"
domain={"Factor"}
range={"Factor"}
```

Properties also have a `datalogRules()` method, which returns a set of pyDatalog-rules for logical inference. For example, for the `isCause` property, the following rule allows to infer new facts `isEffect(X,Y)` from facts `isCause(Y,X)` using an inference engine pyDatalog:

`"isEffect(X,Y) <= isCause(Y,X)"` (1)

here `X`, `Y` – the names of the factors.

2.4. Description of the class Datalog

The `Datalog` class is designed to create agents for logical inference using pyDatalog. The main part of the `rule()` method is:

```
props=getClassObjects('Property')
facts=KBToFacts()
dfacts=self.getDatalogFacts(facts)
drules=self.getDatalogRules()
allFacts=runDatalog(dfacts, drules, props)
factsToKB(allFacts)
```

First, the `rule()` method using `KBToFacts()` retrieves the set of all facts from all property values of all agents as triplets

$$\texttt{(subject, predicate, object)} \tag{2}$$

where `subject` is an agent name, `predicate` is the name of its property, `object` is an element of the set of property values. Then the `getDatalogFacts()` method converts set of facts into the list of pyDatalog-facts with elements `"+predicate ('subject', 'object')"`. The `getDatalogRules()` method generates a set of pyDatalog-rules from the `datalogRules()` methods of all agents. The logical inference of new facts with pyDatalog 0.17.1 is performed in the `runDatalog()` function, which gets lists of all facts, rules and predicates. This function loads the pyDatalog-code with the `load()` function and infers new facts by querying the knowledge base with the `ask()` function for each predicate. New facts are returned in a set of triplets (2). After the logical inference, the `factsToKB()` function returns these facts to the knowledge base `KB` by adding `object` to the set of values of the `predicate` property of the `subject` agent. The `rule()` method is applied until the difference in the sets of new facts and old facts (before rule is applied) is not empty. Inference can also be implemented by other methods. For example, you can easily implement a naive forward chaining algorithm for the rule (1) by adding the following code to the `rule()` method of the `Factor` class:

```
for k in KB:
    if KB[k].__class__.__name__!='Factor': continue
    if k not in self.isCause: continue
    if self.__name__ not in KB[k].__dict__['isEffect']:
        KB[k].__dict__['isEffect'].add(self.__name__)
        return True
```

You can test this algorithm by disabling the `Datalog` agents (`active = False`). But using pyDatalog is more productive.

2.5. Description of the class Fact

The `Fact` class describes the knowledge base fact as a triplet (2). The `source` attribute is the source of the fact. The `rule()` method automatically creates values for agent properties that match this fact. Agent example:

```
"""Fact_stress concentration is cause of fatigue strength decrease"""
subject="stress concentration"
```

```
predicate="isCause"
objecT="fatigue strength decrease"
```

2.6. *Description of the class Model*

The `Model` class describes a finite element model agent of a threaded connection of sucker rods according to GOST 13877-96 (API Spec 11B equivalent). The `paramsIn` and `paramsOut` attributes contain dictionaries with the names and values of the input and output parameters of the model. The `rule()` method creates a new process for simulating a model, passes the `paramsIn` value to it using command line arguments, waits for completion and reads the results from the standard output stream into `paramsOut` using the `parseOutput()` method. If `paramsIn` is empty, the model will be simulated with standard parameter values. The model will be simulated only when among the values of `paramsOut` there is `None`. Agent example:

```
#-*- coding: utf-8 -*-
"""Model_3/4 API Spec 11B_r3n=2.5"""
paramsIn["radiuses on stress relief groove"]=2.5
```

2.7. *Description of the program ThreadsOCC*

For simulation the ThreadsOCC program is used, which is developed by the authors (https://github.com/vkopey/ThreadsOCC). This program is created in the Python language, is a part of the PLM-system and is designed to simulate threaded connections using the finite element method (FEM). Using the program, we can create parametric geometric and axisymmetric finite element models of threaded connections. The boundary representation method, which is implemented by the pythonOCC 0.18.1, is used to create geometric models. CalculiX 2.15 is used to create models for FEM. The program can be used to fully automated optimization of the threaded connection parameters.

ThreadsOCC consists of modules gost13877_96params.py, main.py, ccx_inp.py, ccx_out.py. The module gost13877_96params.py describes the main and additional parameters of threaded connections of sucker rods. The main parameters are nominal values of dimensions with permissible deviations. Additional parameters simplify the creation of the model and depend on the main ones (for example, a point for identifying the edge of a geometric model). Actual values of the parameters are set using `setModelParams()` function. In this module, you can set other model parameters such as material and load parameters.

The main.py module is the main module of the program. It first builds a geometric model, then a finite element model, then simulates and reads results. To simplify the creation of a planar geometric model, functions such as `poly()` (creates a polygon with fillets or chamfers on the vertices), `cut_array()` (creates an array of cuts on the face) and others are developed. Planar geometric models of the nipple (pin), coupling and connection are created using the functions `face1()`, `face2()` and `mkCompaund()`. The created model is saved in the BRep format.

The ccx_inp.py module contains functions for creating a mesh of the FEM-model and input-file for CalculiX. Gmsh 4.4.0 is used to create a mesh from a BRep-file. In the input-file, the program needs to specify the contact conditions, the external load and the boundary conditions on the specified mesh lines. To find these lines in the mesh-file generated by Gmsh, it is parsed (the `ccx_inp.parse()` function) and sets of nodes (`nodes`), lines (`lines`) and elements (`elements`) are obtained. It is also necessary to find the matching between the edges of the geometric model and the mesh lines using such functions as `findEdge()`, `findContEdges()`, `ccx_inp.findLine()`, etc. After that, the final input-file is created and the simulation is executed in CalculiX. Simulation results CalculiX writes to the frd-file.

The ccx_out.py module contains functions for the frd-file parsing, obtaining the stress components at a specified node, calculating the principal stresses, equivalent stress and fatigue safety factor (*FOS*) using the dependence of Sines [25].

The main.py program can read parameter values from command line arguments and write the results to its standard output stream. An example of calculating the minimum *FOS* value in a 3/4 API Spec 11B nipple with parameters *r3n*=2.5, *d_n*=13.12:

```
python main.py r3n=2.5 d_n=13.12
FOS= -13.6997873264
```

2.8. Description of the class Parameter

The `Parameter` class describes threaded connection parameters. Parameter objects have attributes-factors `factorHigh` and `factorLow`, which allow to associate parameters with factors. Parameter example:

```
"""radiuses on the stress relief groove_3/4 API Spec 11B"""
factorHigh="radiuses on the stress relief groove increase"
factorLow="radiuses on the stress relief groove decrease"
```

2.9. Description of the class Dependence

The `Dependence` class describes agents with statistical dependencies of one variable. The attribute `Xpar` is the name of the independent variable, the attribute `Ypar` is the name of the dependent variable. Values `Xpar`, `Ypar` are the keys of the `Parameter` class agents. Attributes `X`, `Y` contain a list of values of independent and dependent variables. The `source` attribute indicates the source of the dependence (reference or model). For example:

```
"""dependence1_3/4 API Spec 11B"""
Xpar="length of stress relief groove"
Ypar="logarithmic fatigue life"
X=[15.0, 25.0, 35.0, 45.0]
Y=[4.3, 5.4, 6.1, 7.0]
source="Kopey V. 2012 Technology audit and production reserves 6/2(8)"
```

X, Y data can be automatically obtained from a set of models (`Model` agents) using the `fromModels()` method, if the `source` is a model and the elements of this set differ from the `source` only by the value of one parameter from `Xpar` (this is verified by the `isSibling()` method from the `Model` class.). The `Dependence` agent can also dynamically create models (agents of the `Model` class) if there are `X` values, some `Y` values are `None` and such a model does not exist. At future iterations, these models will be simulated and the `Y` data will be obtained using the `fromModels()` method. The `rule()` method calls the `fromModels()` and `toModels()` methods and builds linear regression using SciPy. If the value of the coefficient of determination R^2 is high enough (R^2>0.5), then the `rule()` tries to create a new fact:

```
if slope>0: # if positive correlation
    KB[KB[self.Xpar].factorHigh].isCause.add(KB[self.Ypar].factorHigh)
else:
    KB[KB[self.Xpar].factorLow].isCause.add(KB[self.Ypar].factorHigh)
```

Thus, the high correlation between `Xpar` and `Ypar` allows automatically create causal relationships between factors.

2.10. Description of the class DependenceMulti

The `DependenceMulti` class is similar to the `Dependence` class and describes the statistical multivariate dependencies. The attributes `Xpars` and `Ypars` contain lists of the names of dependent and independent parameters. Attributes `X` and `Y` contain their values. Agent description example:

```
"""dependence4_3/4 API Spec 11B"""
Xpars=["radiuses on stress relief groove",
       "major radius of nipple thread "]
X=[[00.5, 01.5, 02.5, 03.5, 00.50, 01.50, 02.50, 03.50],
   [12.7, 12.7, 12.7, 12.7, 13.26, 13.26, 13.26, 13.26]]
Ypars=["fatigue safety factor"]
Y=[[None,None,None,None,None,None,None,None]]
source="3/4 API Spec 11B"
```

For processing and analyzing `X` and `Y` data, you can use the pandas package and its `DataFrame` data structure. For two-way conversion to `DataFrame`, the `toDataFrame()` and `fromDataFrame()` methods are used. The `linregress()` method is used to build a linear regression using the scikit-learn package and returns its coefficients and coefficient of determination. If the source is a model, and some elements of `Y` are unknown (are `None`), then the `byModels()` method allows to automatically obtain the `Y` values by creating temporary models and simulating them. The `optimize()` method allows you to find the optimal values of `X` according to the specified criterion `yp` by simulating models and applying algorithms for multi-variable function optimization from SciPy. This method optimizes the values that the `simModel(x, yp)` method returns, where `x` is the argument vector, `yp` is the name of the dependent parameter. The following example shows the use of the `optimize()` method:

```
d=KB[dependence4]
d.optimize(yp="fatigue safety factor", bounds=[(2.5,3.5),(12.7, 13.26)],
maximize=True)
```

Unlike the `Dependence` class, automated creation of causal relationships between the factors from the correlation of `Xpars` and `Ypars` is not implemented. This must be done by the user.

2.11. An example of interaction with the system

The system supports interactive work in Python or IPython shells. To simplify code editing, authors recommend using a python-IDE with an integrated IPython-shell. To facilitate the work with the system (creating and editing agents, creating queries and visualization) additional modules with GUI can be developed. Below is an example of interaction with the system in Python-shell:

```
>>> from tools import * # import everything from the module
>>> files=getFiles() # get the list of modules from the current directory
>>> createClasses(files) # create agent classes
>>> createKB(files) # create agents
>>> applyRules() # apply the rules as long as there are changes
>>> saveKB() # save agents
>>> loadKB() # load agents
```

The following query prints all causes of the `"stress concentration"` factor:

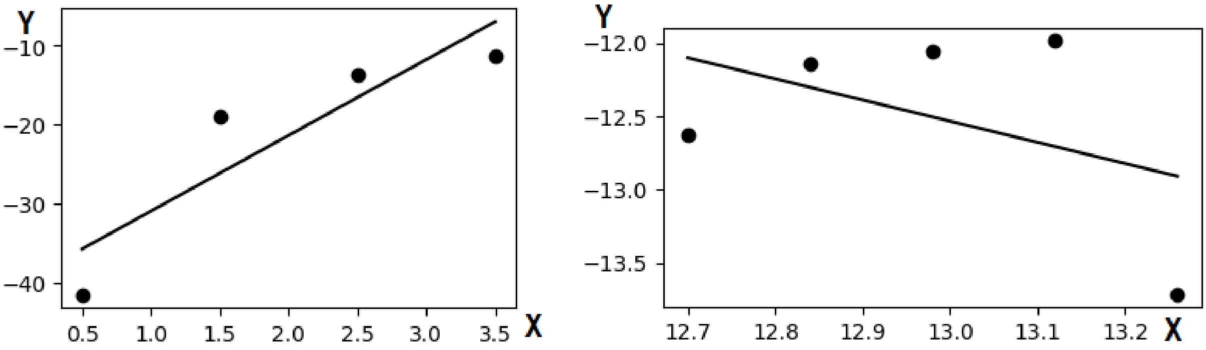
```
>>> for v in KB["stress concentration"].isEffect:
```

```
...       print v
corrosion damage
corrosion pit
```

The following query builds the `"dependence2"` dependence and its linear regression using Matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> d=KB["dependence2"]
>>> d.linregress()
LinregressResult(slope=9.59, intercept=-40.58, rvalue=0.9, pvalue=0.1,
stderr=3.35)
>>> d.plot(plt)
```

Dependencies in Figure 1 were obtained automatically from the `Model` agents (3/4 API Spec 11B connections). According to the dependence in Figure 1(a), the system infers fact `("radiuses on stress relief groove increase", "isCause", "fatigue strength increase")`. Such inference from dependence in Figure 1(b) is not made due to the small value of R^2.



a b

Figure 1. Dependences of *FOS* (*Y*) on geometric parameters (*X*), mm: radiuses on the stress relief groove, R^2=0.8 (a); major radius of the nipple thread, R^2=0.19 (b)

The following query builds a directed graph using NetworkX with `"isCause"` edges and `Factor` class vertices and calculates PageRank [26] values for them based on the structure of the incoming links. These PageRank values can be used to evaluate the importance of a factor — the relative number of factors causing this factor. A graph with PageRank values is visualized (Figure 2) using Graphviz by converting it into the graph description language (DOT) [27].

```
import networkx as nx
G = nx.DiGraph()
for i in KB:
    if KB[i].__class__.__name__=="Factor":
        for j in KB[i].isCause:
            G.add_edge(i, j, label="isCause")
pr=nx.pagerank(G)
```

Figure 3 shows the graph that is the result of a query to the `KB` and contains vertices — agents of the classes `Factor`, `Fact`, `Parameter`, `Dependence`, and edges — the properties `isEffect`, `subClassOf`, `isContraryOf`, `objecT`, `subject`, `factorHigh`, `factorLow`, `Xpar`, `Ypar`, `source`.

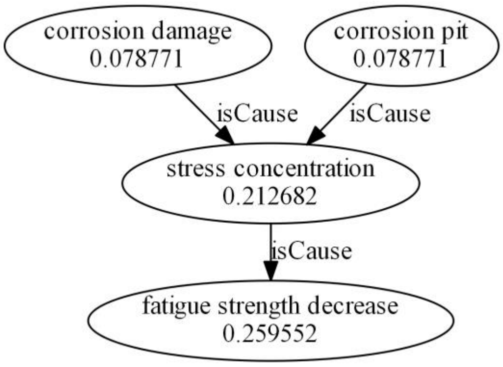


Figure 2. Directed graph of factors with PageRank values (part of the graph is shown)

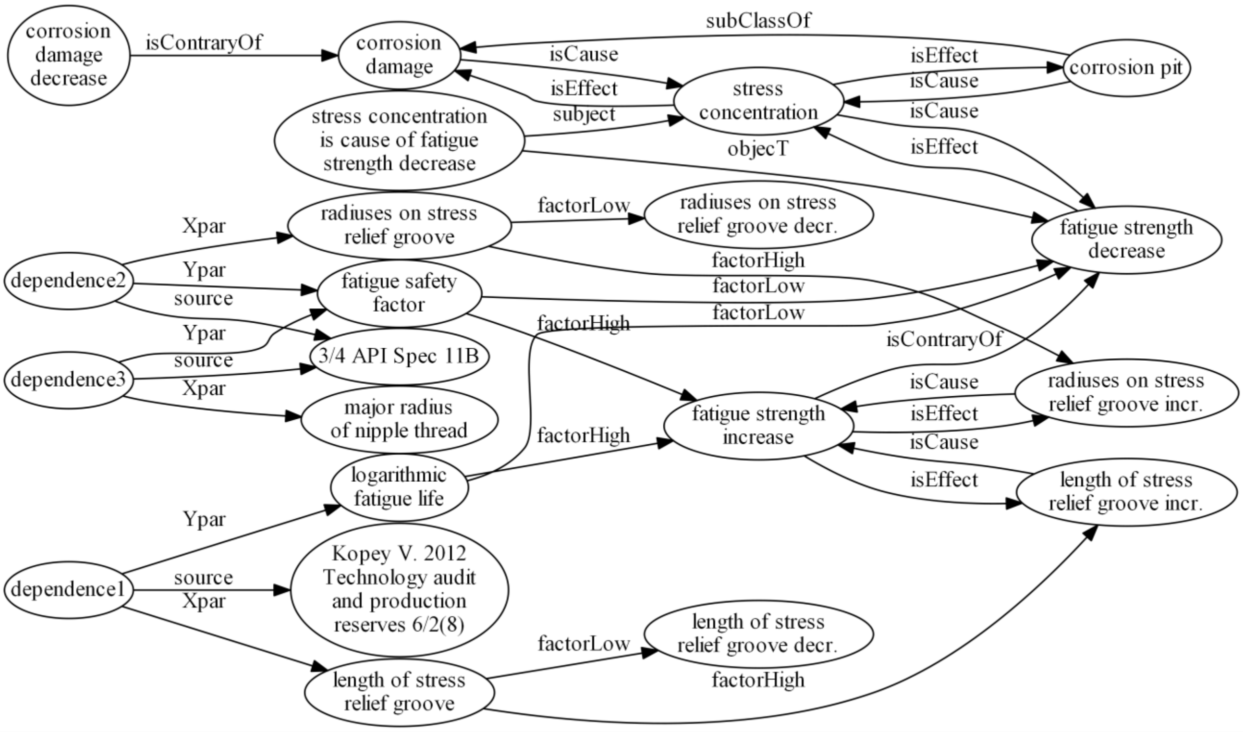

corrosion damage decrease
isContraryOf
corrosion damage
subClassOf
isCause
isEffect
subject
stress concentration
isEffect
isCause
isCause
isEffect
corrosion pit
stress concentration is cause of fatigue strength decrease
objecT
fatigue strength decrease
Xpar
radiuses on stress relief groove
factorLow
radiuses on stress relief groove decr.
factorHigh
dependence2
Ypar
source
fatigue safety factor
factorLow
factorLow
factorHigh
isContraryOf
Ypar
source
Xpar
dependence3
3/4 API Spec 11B
major radius of nipple thread
isCause
isEffect
isCause
isEffect
fatigue strength increase
radiuses on stress relief groove incr.
factorHigh
logarithmic fatigue life
length of stress relief groove incr.
Ypar
dependence1
source
Xpar
Kopey V. 2012 Technology audit and production reserves 6/2(8)
factorLow
length of stress relief groove decr.
factorHigh
length of stress relief groove

Figure 3. Visualization of the query to the knowledge base

The paper demonstrates examples for the design phase of a threaded connection, but the system can also be completed with agents for other phases of the product life cycle. For example, you can integrate a model for a geometric simulation of thread machining [28]. The issues of system performance or building a distributed system were not considered, but this can be implemented in the future using well-known methods and Python-packages [21-23].

3. Conclusions

The proposed principles for the development of PLM-systems create prerequisites for the appearance of the properties of complex systems in them and, thus, increase their effectiveness. An example of a PLM-system for threaded connections of oil and gas equipment is given, which:

- aims at simple and efficient integration of many heterogeneous components and belongs to the class of multi-agent systems and hybrid intelligent systems;

- focuses on a simple description of agents, their behavior and interactions, the dynamic creation and persistence of agents, the extension of the functionality of the system by creating new classes and agents;
- developed by the popular Python language, allows you to work with the system in the Python-shell, uses various open-source Python-packages and third-party software, including the pyDatalog inference engine and a program developed by the authors for finite element simulation of threaded connections.

References

[1] Saaksvuori A and Immonen A 2008 *Product lifecycle management*, Third Edition, Springer-Verlag, Berlin, Heidelberg
[2] Kopey V B 2017 Abstract model of product lifecycle management system, *Precarpathian bulletin of the Shevchenko scientific society* **2**(38) 71-96 (in Ukrainian)
[3] Bertalanffy von L 1968 *General system theory: foundations, development, applications*, George Braziller, Inc., New York
[4] Kossiakoff A, Sweet W N, Seymour S J and Biemer S M 2011 *Systems engineering principles and practice*, Second Edition, A John Wiley & Sons, Hoboken, New Jersey
[5] Volkova V N 2006 *Gradual formalization of decision-making models*, Saint Petersburg State Technical University, Saint Petersburg (in Russian)
[6] Gavrilov A V 2002 *Hybrid intelligent systems*, Novosibirsk State Technical University, Novosibirsk (in Russian)
[7] Shoham Y and Leyton-Brown K 2008 *Multiagent systems: algorithmic, game-theoretic, and logical foundations*, Cambridge University Press
[8] Kravari K and Bassiliades N 2015 A survey of agent platforms, *Journal of Artificial Societies and Social Simulation* **18**(1) 11
[9] Shoham Y 1993 Agent-oriented programming, *Artificial Intelligence* **60** 51-92
[10] Karasev V O and Sukhanov V A 2017 Product lifecycle management using multi-agent systems models, *Procedia Computer Science* **103** 142-147
[11] Fortineau V, Paviot T and Lamouri S 2019 Automated business rules and requirements to enrich product-centric information *Computers in Industry* **104** 22-33
[12] Paviot T *PythonOCC, 3D CAD/CAE/PLM development framework for the Python programming language* http://www.pythonocc.org (accessed 2019-05-01)
[13] Jones E, Oliphant E, Peterson P, et al. *SciPy: open source scientific tools for Python* http://www.scipy.org (accessed 2019-05-01)
[14] Hunter J D 2007 Matplotlib: A 2D graphics environment, *Computing in Science & Engineering* **9**(3) 90-95
[15] Pedregosa F, et al. 2011 Scikit-learn: machine learning in Python, *JMLR* **12** 2825-2830
[16] *pyDatalog* https://sites.google.com/site/pydatalog (accessed 2019-05-01)
[17] McKinney W 2010 Data structures for statistical computing in Python, *Proc. of the 9th Python in Science Conf.* ed S van der Walt and J Millman, pp 51-56
[18] Hagberg A A, Schult D A and Swart P J 2008 Exploring network structure, dynamics, and function using NetworkX *Proc. of the 7th Python in Science Conf. (SciPy2008)* ed Gäel Varoquaux, Travis Vaught et al., Pasadena, CA USA, Aug 2008, pp 11-15
[19] Dhondt G 2004 *The finite element method for three-dimensional thermomechanical applications,* Wiley
[20] Geuzaine C and Remacle J F 2009 Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities, *International Journal for Numerical Methods in Engineering* **79**(11) 1309-1331
[21] Ettienne M B, Vester S and Villadsen J 2012 Implementing a multi-agent system in Python with an auction-based agreement approach, *Programming Multi-Agent Systems. ProMAS 2011. Lecture Notes in Computer Science*, ed Dennis L, Boissier O et al., **7217** Springer, Berlin

[22] ****osBrain – A general-purpose multi-agent system module written in Python* https://github.com/opensistemas-hub/osbrain (accessed 2019-05-01)
[23] ****Python Agent DEvelopment framework* https://pade.readthedocs.io (accessed 2019-05-01)
[24] McKerns M M, Strand L, Sullivan T, Fang A and Aivazis M A G 2011 Building a framework for predictive science, *Proc. of the 10th Python in Science Conf.* http://arxiv.org/pdf/1202.1056
[25] Sines G 1959 Behavior of metals under complex static and alternating stresses, *Metal Fatigue* eds G Sines and J L Waisman, McGraw-Hill, New York, pp 145-169
[26] Page L, Brin S, Motwani R and Winograd T 1999 *The PageRank citation ranking: bringing order to the Web* http://dbpubs.stanford.edu:8090/pub/showDoc.Fulltext?lang=en&doc=1999-66&format=pdf
[27] Gansner E R and North S C 2000 An open graph visualization system and its applications to software engineering *Software-practice and experience* **30**(11) 1203-1233
[28] Kopei V B, Onysko O R and Panchuk V G 2019 Computerized system based on FreeCAD for geometric simulation of the oil and gas equipment thread turning, *IOP Conf. Ser.: Mater. Sci. Eng.* **477** 012032