UDK 004.052.32

J. Fiser, PhD.,
V. Mashkov, ScD., **V. Lytvynenko**, ScD.

REPRESENTATION OF SYSTEM LEVEL SELF-DIAGNOSIS IN PYTHON PROGRAMMING LANGUAGE

Abstract. *This paper describes principles of implementation of basic entities on system level of self-diagnosis (especially self-checking) in Python programming languages. The first part discusses usability of Python for representation and simulation of complex systems (comparing it primarily with the most important competitor in the field of universal programming languages – Java programming language).The second part depicts main principles of implementation of basic entities of system level self-diagnosis by real examples including calculating of basics characteristics of a system. The proposed representation forms the basis of event-based simulation of more complex systems.*

Keywords: *self-diagnosis, complex systems, Python, simulation*

Й. Фишер, канд. техн. наук,
В. А. Машков, **В. И. Литвиненко**, доктора техн. наук

ПРИМЕНЕНИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ PYTHON ДЛЯ РЕШЕНИЯ ЗАДАЧ САМОДИАГНОСТИКИ НА СИСТЕМНОМ УРОВНЕ

Аннотация. *Описаны принципы реализации базовые принципы организации решения задачи самодиагностики с применением языка программирования Python. В первой части рассмотрены преимущества применения языка Python для представления и моделирования сложных систем по сравнении с языком программирования Java. Во второй части статьи рассмотрены основные принципы реализации базовых субъектов на системном уровне самодиагностики с использованием реальных примеров, включая расчет основных характеристик системы. Предложенный подход лежит в основе моделирования на основе событий применительно к более сложным системам.*

Ключевые слова*: самодиагностика, сложная система, Python, моделирование*

Ї. Фішер, канд. техн.наук**,**
В. А. Машков, **В. І. Литвиненко**, доктора техн. наук

ЗАСТОСУВАННЯ МОВИ ПРОГРАМУВАННЯ PYTHON ДЛЯ ВИРІШЕННЯ ЗАВДАНЬ САМОДІАГНОСТИКИ НА СИСТЕМНОМУ РІВНІ

Анотація. *Описано принципи реалізації базові принципи організації вирішення задачі самод-гностики із застосуванням мови програмування Python. У першій частині розглянуті переваги застосування мови Python для представлення та моделювання складних систем по порівнянні з мовою программирования Java. У другій частині статті розглянуті основні принципи реалізації базових суб'єктів на системному рівні самодіагностики з використанням реальних прикладів, включаючи розрахунок основних характеристик системи. Запропонований підхід лежить в основі моделювання на основі подію-тий стосовно до більш складним системам.*

Ключові слова: *самодіагностика, складна система, Python, моделювання*

1. System level self-diagnosis

Research in the field of system level diagnosis was introduced in 1960 s by Franco Preparata P. [1]. System diagnosis aims at diagnosing systems composed of modules (units) with the requirement that they are able to test each other by exchanging information through available links [2]. At system level, each particular exchanging is considered as atomic test, which return value 0 or 1. The sets of test results is called syndrome.

In the most elementary case, the self-diagnosis system is defined only by three structures:

• set of permanent states of modules;

• directed graph of atomic tests (test are processed only once);

• syndrome (= results of all atomic tests assuming that fault free unit detects faults in all tested, units i.e. 100 % fault coverage).

This type of system can be represented by weighted graph (nodes representing modules are marked by state and edges are marked by results of atomic tests), see Fig. 1 on the following page.

The more complex systems relax some limitation, e.g. permanent character of faults, absolute fault coverage etc. Moreover, the advanced systems have to provide internal processing of syndrome, online diagnosis of global state of system and elimination of global failures.

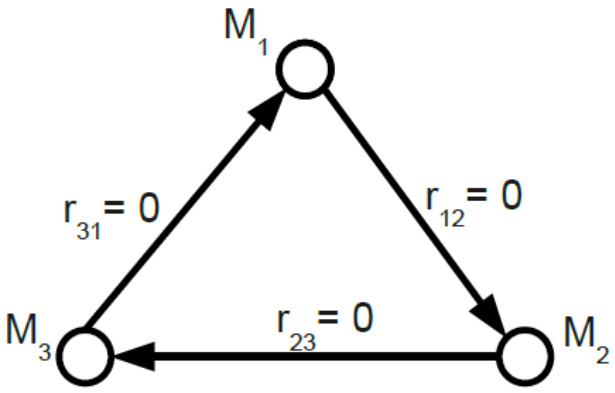


Fig. 1. Diagnostic graph with syndrome

2. Programming language for simulation ofsystem level self-diagnosis

Essential requirements for programming languages, which might be useful for a simulation of system level self-diagnosis:

1) **simple data representation** of complex models of modules and networks of atomic checks (simulation are often data-oriented or data-centric);

2) **effective representation of n-dimensional data spaces** (typical representation of working data);

3) support of **task-oriented parallelism** (simulations problems have time complexity O(n2) or even worse);

4) customizable graphical output for **directed graphs and 2D or 3D graphs of functions;**

5) embedded support of **event-based simulations** or a high level simulation library;

6) support of **symbolic evaluations.**

Other criteria are more subjective, but they have influenced our choice:

Long-term sustainability – the language (including supporting libraries) must be stable with a long term support. Prediction of future support is difficult, but it should take into account at least history of product and activity of its supporters.

Support of classical imperative paradigm with an simple and ordinary syntax – our students and especially staff have limited experience with more exotic paradigmatic and syntactic approaches (e.g. pure functional languages). However, support of others paradigmatic is appreciated.

Multiplatform support – our students and staff use both Linux and MS Windows

Open-sourceness – low cost and the opportunity to study library sources

No programming language meets these criteria completely. The best match provides Java and Python. Both languages are mature, multiplatform, based on imperative paradigm and open-source (including necessary auxiliary libraries).

Python programming language natively support functional idioms which make some algorithms more compact and self-descriptive (e.g. summation). Java supports some basic functional headstones in the newest version (Java 8) but this support is limited (list comprehensions are the main missed feature).

Both languages are primarily based on the object oriented paradigm. Therefore, the representation of complex data structures are straightforward and flexible (all entities of self-diagnosis system are representable as objects). The OOP semantic and syntax of Python is simpler and easier to understand for non-OOP programmers. On the other hand it is more modifiable and flexible by mean of meta-object protocol.

The support of parallelism in Java and Python is comparable. The threading in Java, and multiprocessing in Python makes possible coarse-grained parallelism by mean of high-level constructs and entities (e.g. futures or queued tasks). The Python standard multiprocessing library additionally supports a light-weight distributed computing over TCP/IP.

Java is popular language for event-based simulation from its beginnings. There are a lot of Java simulation frameworks. We have tested DESMO-J framework [3] (this framework is directly linked with classical *Simula* 68). In Python, only one extensive framework exists – *SimPy* (simpy.readthedocs.org). On the other hand, Python supports cooperative coroutines directly in language (via generators). The representation of process-based simulation by coroutine is more effective than representation by expansive natural threads (natural threads use a lot of system resources e.g. kernel memory).

The support is not so balanced in the field of graphical output. Java provides several plotting libraries which are focused on plotting of 2D function graphs e.g. JFreeChart. In Python world, the matplotlib library is de facto standard for 2D plotting [4] (matplotlib products more journal-friendly diagram than JFreeChart). Support of 3-D

graphics in Java and Python is more limited (as compared with specialized proprietary software packages as *Matlab* or *Mathematica*).

The most problematic and distinctive feature is rendering of directed-graph. Both languages are used in many digraph supporting libraries, but these libraries are often only school exercises or unstable and unsupported alpha versions. The output is commonly provided by external tools (e.g. graphic) or it use simple force-directed (spring) algorithm.

However, Pythonworld offers *NetworkX* framework (networkx.github.io) which provide a rich set of graph representations and algorithms and relatively customizable drawing procedures (all basic drawing algorithms are supported including force based, eigenvector spectrum, concentric shells, &c.).

The crucial difference between Python and Java (for our point of view) is embedded support of compact numeric multidimensional arrays on the one hand and a symbolic computing on the other one.

The Java support of multidimensional arrays is only on the basic level. Vector (one or multidimensional) must be represented by multilevel referenced structures (noncompact with slow access) and vector based operation are not supported in language syntax (i.e. arrays are not first class objects).

The Python support of multidimensional arrays on basic language level is similar. *Pythonic* lists have analogous representation and limitations (including a rudimentary support of vector operations). However, the *NumPy* framework [5] extends this support considerably. *NumPy*makes possible compact representation of vector data, vector operations (with natural infix operators), extended indexing, importing and exporting from/to standard format (CSV, HDF5). The vectorized operations are implemented by C (via *Cython*) and supports *MIMD* multiprocessing (via *OpenMPI*) and *SIMD* units (*SSE3*).

Symbolic computing (manipulation with symbolic expression) in Java is restricted by strict static typing of the language and by limited support of operator overloading (i.e. symbolic computing is possible but uncomfortable).

The Python is language with dynamic typing together with perfect operation overloading allows almost natural representation of symbolic computations (unfortunately, Python do not support plain i.e. unbound symbols). The basic support of symbolic expressions provides *SymPy*(*www.sympy.org*) framework. More advanced support is offered by *Sage framework* [6]. This framework use slightly modified version of older Python (version 2.7) but there is a possibility of interoperability with pure Python 3 (our preferred version). Sage is project which combines under one roof several Python mathoriented frameworks (including aforementioned *NetworkX*, *Matplotlib*, *SimPy* and many others) with unified interface and graphical shell (notebook) and it is trying to compete with established proprietary numerical computing environments (as *Matlab*, *Mathematica* or *Maple*).

3. Main building of implementation

The central entity of our OOP representation of a system level self-diagnosis is the class of modules which performs elementary checks (self-checking) and self-diagnosis based on syndrome. The concrete implementation has to extends abstract class Module.

```
Class MState(Enum):
  '''enumeration of module states'''
  OK = 0
  FAULTY = 1
class Module:
def __init__(self, ident, initial-
State=MState.OK):
self.state = initialState
  self.id = ident
  @property
def faulty(self):
    '''abbreviated notation (property ac-
cessor) for test of faulty state'''
returnself.state == MState.FAULTY
defatomicControlOn(self, otherModule):
    '''atomic control (elementary check)
provided by 'self' module on 'otherModule' '''
raiseNotImplementedError("abstract method")
defprobabilityOfResult(self, otherModule,
result,
selfState=None, otherState=None):
    '''
returnprobbality of 'result' (0 or 1) that is
provided by 'self' module
on 'otherModule' assuming states 'self-
State' and 'otherState' of modules.
    '''
```

```
raiseNotImplementedError("abstract method")
def __rshift__(self, otherModule):
    '''
syntactic sugar, in_x (operator like) notation of elementary checks
        a >> b in place of m.atomicControlOn(b).
    '''
returnself.atomicControlOn(otherModule)
def __str__(self):
    ''' only for debugging purposes '''
return self.id + "(" + str(self.state) + ")"
```

Modules – instances of concrete classes are identified by internal identifiers (internal identifier is independent on containing systems), they have initial state, perform atomic checks and compute conditional probabilities of results.

System modules are implemented as container of modules with an added functionality.

```
class System:
def __init__(self, modules, dgraph):
    '''
    'modules' is iterator over modules
    'dgraph' (diagnostic graph) is list of pairs of indices
    '''
self.moduleList = list(modules) # ordering of modules
self.modules = {m.id: m for m in self.moduleList} # dictionary of modulse
self.size = len(self.modules)
    # representationod diagnostic graph - - - set of instances of class 'AtomicCheck'
self.dg = {AtomicCheck(m_i=self.moduleList[i],
m_j=self.moduleList[j]) for i, j in graph}
defgetSyndrome(self):
    '''
return syndrome after preforming of all atomic checks
    '''
    s = Syndrome(self)
for (m_i, m_j) in self.dg:
s.addResult((m_i, m_j), m_i>>m_j)
return s
deftoAdjacencyMatrix(self):
    '''
return system and its diagnostics graph as adjacency matrix (represented by NumPy array).
```

Adjacency matrix is input for some graph algorithm and transitional representation for external tools (e.g. plotting)

```
    '''
matrix = np.matrix(np.zeros((self.size, self.size), dtype=np.int8), copy=False)
for (m_i, m_j) in self.dg:
matrix[self.mpos(m_i), self.mpos(m_j)] = 1
return matrix
# access to modules
def__iter__(self):
    """return iterator over all modules"""
returniter(self.moduleList)
defmpos(self, module):
    '''return index of module in the system (external identi_er)'''
returnself.moduleList.index(module)
def __getitem__(self, index):
    '''indexation (return module for given index)'''
returnself.moduleList[index]
@staticmethod
defgenerateModules(size, moduleClass, faultyModules):
    '''
Factory method which generates iterator over modules (usable in constructor).
The concrete class of modules is transfered via 'moduleClass' argument. Modules
from 'faultyModules' has initial state 'faulty'.
    '''
return (moduleClass(str(i),
MState.FAULTY if i in faultyModules else MState.OK)
foriin range(1, size+1))
# property returning basic characteristic of systems
defATCount(self):
    '''number of elementary checks'''
return sum(np.nditer(self.toAdjacencyMatrix()))
def T(self):
    '''number of faulty modules'''
return sum(1 if module.faulty else 0 for module in self.moduleList)
defTmax(self):
    '''maximal number of faulty modules'''
return (self.size - 1) // 2.
```

Modules in the systems are identified both by their internal identifiers and by indices in ordered list of modules. This allows implementation of algorithms with prevalent module-centric point of view (e.g. local based diagnosis) or algorithms on whole system, which prefer simple indexing. The dual representation is used also for graph of atomic checks. Primary representation is a linked structure (modules are interlinked by atomic checks objects), which are more flexible (atomic checks can carry additional attributes). Derived (computed) representation by adjacency matrix is more compact and portable.

4. Characteristics of diagnostics graph

The computations of characteristics of a diagnostics graph inside the class System are trivial. However, the design of basic classes makes easier calculation of more complex characteristics.

As example, we present calculation of aposteriory probability of hypothesis about global state of system considering obtained syndrome R [7]. The hypothesis is defined by sequence of states of modules $S_0, S_1, S_2, \ldots, S_n$ where $S \in 0,1$ (1 represent faulty module).

The aposteriory probability is calculated by this Bayesian equation:

$$P(H_i / R) = \frac{P(H_i) \cdot P(R / H_i)}{\sum_{j=0}^{2^{n-1}} \left(P(H_j) \cdot P(R / H_j) \right)}$$

where n is number of modules, H_i is hypothesis with sequence of states which forms n-digit binary representation of number $i \left(i \in 0 \ldots 2^n - 1 \right)$.

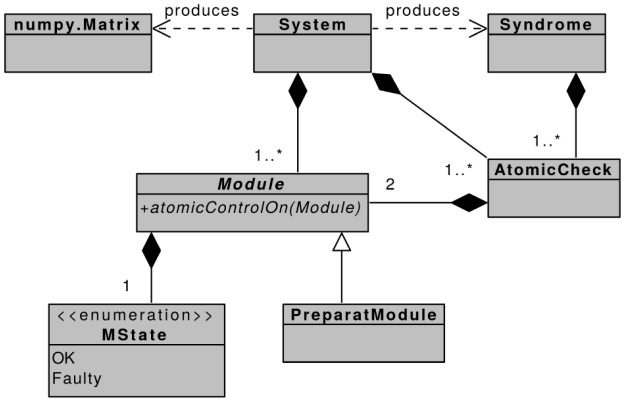


Fig. 2. UML class diagram of basic classes

The implementation is based on new class *Hypothesis* that encapsulate given states of modules in one integer number (Python uses integers with arbitrary-precision arithmetic; therefore integer number is able to represent bit vectors with arbitrary length).

This class provides basic initializes, accusers and iterators over all possible hypotheses. This iterator is represented by the generator (= coroutine producing iterator [8]) which makes next object of hypothesis only in the moment of a request (lazy evaluation). This approach makes iteration memory very effective (the array of 2^n 2n integer items is never created).

```
defproduct(iterator):
    '''product of iterator over numbers'''
return reduce(operator.mul, iterator)
class Hypothesis:
def __init__(self, size, value):
self.size = size
self.max = 2**size
self.value = value
def __getitem__(self, i):
    ''' return hypothetical state of module 'i' for 'self' hypothesis)'''
return(self.value>>i) & 1
@staticmethod
defh0(size):
    '''
    factory method for hypothesis H_0 (all modules are faulty free)
    '''
return Hypothesis(size, 0)
defnextHypothesis(self):
    '''
    return object of hypothesis H_n+1
    '''
return Hypothesis(self.size, (self.value + 1) % self.max)
def__eq__(self, other):
    '''equality for hypotheses'''
returnself.value == other.valueand self.size == other.size
def__ne__(self, other):
    '''inequality for hypotheses'''
returnself.value != other.value or self.size != other.size
defgenAllHypotheses(self):
    '''
    generator producing iterator over all hypotheses (beginning from 'self' hypothesis).
    '''
```

```
    yield self
    nh = self.nextHypothesis()
    whilenh != self:
    yieldnh
    nh = nh.nextHypothesis()
    defapost_probability(self, syndrome):
      '''
    aposteriory probability of 'self' hypothesis depending on obtained syndrome
    input:
          1) hypothesis (self)
          2) syndrome as result of self-checking in a system
          3) apriory probabilities of module failures (including in module object in the system)
      '''
      # basic precondition of compatibility of hypothesis and syndrome
    assertself.size == syndrome.system.size
      s = syndrome.system # system of syndrome
    suma = 0.0
    nom = None
      # loop over all hypotheses (beginning from H_0)
    for h in Hypothesis.h0(self.size).genAllHypotheses():
        # apriori probability of hypothesis 'h'
    ph = product(s[i].P_m if h[i] == 0 else 1-s[i].P_m
    foriin range(self.size))
        # probability P(R/H) of syndrome for hypothesis 'h'
    prh = product(m_i.probabilityOfResult(m_j, r,
    selfState=MState(h[s.mpos(m_i)]),
    otherState=MState(h[s.mpos(m_j)]))
    for (m_i, m_j, r) in syndrome)
        # sumation of denominator
    suma += ph * prh
    if h == self:
        # nominator (probability for 'self' hypothesis)
    nom = ph * prh
    return nom/suma
```

The most striking feature of this Python code is utilization of generator expressions [9], which together with function product mimics mathematical *product* operator (generator expression are argument of this function). Moreover, the generator expression used over iterator products lazy iterator again (i.e. the memory efficiency is preserved).

The loop over all hypotheses is relatively easily transformable to parallel version by multiprocessing library. The loop has to be replaced by generator expression as argument of method *imap* of task pool that in parallel map evaluation of $P(H_i)\cdot P(R/H_j)$ over hypotheses. Resulting iterator could be (persisting laziness) summed by standard function *sum*. Unfortunately, speed-up achievable by mean of parallelization ($i < n$ - times, where n is the number of computing cores) is insignificant for the task with exponential time complexity $O(2^n)$. For system with more than 25 modules computing time is in unit of days (single-threaded implementation).

5. Conclusions

The pythonic representation of self-diagnosis entities which are briefly described in this article is headstone for more complex simulations, which make possible research in the field of systems with intermittent failures and especially in most challenging area of self-diagnosis (i.e. diagnosis without external device).

These simulations requires framework for event/process-based simulations with support of process coroutines, shared resources and inter-process signaling. We use SimPy as low-level foundation for more complex and flexible framework ETOS. This framework provides mechanism for creation of simulation processes from instances of relatively simply classes (denoted as activities). The high-level description of process flow is represented by XML.

The interconnection of ETOS activities and classes of our library of self-checking primitives is possible by mechanism of mixins class [10] which is based on (secure) multiple inheritance. The representations of object behavior combine functionality of abstract simulations activities (ETOS) with abilities of self-diagnosis classes (extended versions of aforementioned classes).

References

1. Preparata F.P., Metze G., and Chien R.T, R.T. (1967), On the Connection Assignment Problem of Diagnosable Systems, *IEEE*

Transactions on Computers, Vol. EC-16, No. 6, pp. 848 – 854, http://dx.doi.org/10.1109/PGEC.1967.264748.

2. Mashkov V., (2011), Selected Problems of System Level Self-diagnosis, *Ukrainian Academic Press,* Lvov, Ukraine.

3. Göbel J., Joschko P., Koors A, and Page B., (2013), The Discrete Event Simulation Framework DESMOJ: Review, Comparison to other Frameworks and Latest Development, *In Proceedings of the 27th European Conference on Modeling and Simulation,* Ålesund, Norway.

4. Hunter J.D., (2007), Matplotlib: A 2D graphics Environment, *Computing In Science & Engineering,* Vol. 9, No. 3, pp. 90 – 95.

5. David Ascher, Paul F. Dubois, Konrad Hinsen, James Hugunin, and Travis Oliphant, (1999), Numerical Python. Lawrence Livermore National Laboratory, Livermore, CA, ucrlma-128569 edition.

6. Stein W.A. et al., (2015), Sage Mathematics Software (Version 6.4). The Sage Development Team, available at: http://www.sagemath.org (accessed 24.03.2015).

7. Fiser J., and Mashkov V., (2010), Generic Model for Application of Probabilistic Algorithms for System Self-diagnosis, *Proceedings of ISDMCI'2010 International Conference*, Yevpatoria, Ukraine, pp. 215 – 218.

8. Schemenauer N., Peters T., and Hetland M.L., (2001), PEP 255 – Simple Generators, *Tech. Rep*., May 2001, available at: https://www.python.org/dev/peps/pep-0255 (accessed 24.03.2015).

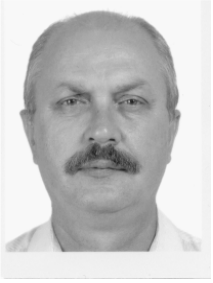
10. Raymond H., (2002), PEP 289 – Generator Expressions, *Tech. Rep*., Jan 2002, available at: tt https://www.python.org/dev/peps/pep-0289 (accessed 24.03.2015).

11. Esterbrook C., (2001), Using Mix-ins with Python, *Linux Journal*, Apr 2001, available at: http://www.linuxjournal.com/article/4540 (accessed 24.03.2015).

Received 28.02.12015

Fiser
Jiri, PhD., Senior lecturer of informatics University of J.e. Purkinje 400 01, Usti nad Labem, Ceske mladeze 8, Czech Republic.
T. +420 47528 3902.
E-mail: jf@jf.cz

Mashkov
Viktor, Doctor of Science in Engineering Docent Department of IT at the University of J.E. Purkyne in Usti nad Labem (Czech Republic). Ceske mladeze 8, 40096 Usti nad Labem Czech Republic.
T. +420 475 283 908.
E-mail: viktor.mashkov@ujep.cz



Lytvynenko
Volodymyr, Doctor of Sciene, Department Informatics and Computer Science. Head of Department, Kherson National Technical University.
T.+380 50 9955 202.
73008, Kherson, Berislavsky Shosse, 24, Ukraine.
E-mail:
immun56@gmail.com