CVE (NVD) Ontology Generator

Vladimir Dimitrov [1]

[1] *University of Sofia, 5 James Bourchier Blvd., Sofia, 1164, Bulgaria*

Abstract

NVD is a very huge database. The corresponding ontology generated from this database therefore is huge one. An ontology to be manageable by humans must be organized in adequate way.

The biggest issues in NVD ontology generation are ontology structuring and performance.

Generation of NVD ontology originally continues in 2–3 days. Maximum parallelism must be used to short this execution time.

NVD ontology structuring must be done in a way permitting only parts of the whole ontology to be used.

These issues and the solution are discussed is this paper.

Keywords

CVE, ontology, vulnerabilities, Semantic Web

1. Input data and organization

NIST provides data on zip-compressed vulnerabilities in JSON format. From 2002 until now, there is a separate file for each year with vulnerabilities registered in that year. There is also an additional file nvdcve-1.1-modified.json.zip, which contains the latest changes made to vulnerabilities from previous years. This is done in order not to have to re-download the files in which changes should be made. In practice, the last file can be used as a patch on the contents of files from previous years.

There is another file nvdcve-1.1-recent.json.zip, which contains the latest registered vulnerabilities, but the information from it is duplicated with that of the current year's file.

The description of the ontology used in the generation can be found in the shell.owl file, and the descriptions of the ontology used in the generation of the individual parts of the ontology can be found in the shellPart.owl.

There is another file for generating shellConfig.owl configurations.

The size of the whole ontology are enormous. Now, a complete generation of the ontology has not been made, but only in separate parts years.

Information Systems & Grid Technologies: Fifteenth International Conference ISGT'2022, May 27–28, 2022, Sofia, Bulgaria
EMAIL: cht@fmi.uni-sofia.bg (V. Dimitrov)
ORCID: 0000-0002-7441-253X (V. Dimitrov)

The reason for these ontology volume related to the size of the configurations (platforms) to which the individual vulnerabilities relate.

The decision is fully to deploy CPE expression matches to base CPE names is because RDF is a static graph. It does not execute requests to track the links. In NVD, vulnerabilities are not directly indicated in the base CPE names [1], but CPE expression expressions from the CPE language [2] are most commonly used. Thus, NIST maintains the database compressed. This approach creates problems when using the NVD database by external users and therefore NIST provides another file nvdcpematch-1.0.json.zip, which contains these queries (CPE compliance expressions) to the base CPU names.

A separate ontology is generated for each year, which can be presented independently and integrated by inclusion in the full ontology.

On the other hand, most of the years, ontologies are also huge and cannot be viewed with normal text editors or such specialized for ontologies. This necessitated the breakdown of the ontology for a given year into parts that can be presented (and used) independently in the Protégé [3]. These parts for the year are combined in one ontology for the respective year.

These breakdowns of the ontology are not enough – the configurations for the vulnerabilities are also very huge. In fact, the configurations are exported in separate files.

Finally, for each part of the ontology of a given year, the configurations of the vulnerabilities contained in it are exported in a separate series of files.

The ontology for a given part of the year can be considered independently because it includes all its configurations (files).

2. The implementation

The generator code is in the generateNVDontology.py file.

This program can be started with parameters. The first one is for downloading files from the NIST website, and the second – for which years to generate the ontology.

Copies of the files from the NIST site are stored in the “data” subdirectory. Along with the files themselves, their meta-files are also downloaded, which in particular contain information about the date and time of generation of the respective file by NVD.

The operation of the generator `main(download, feeds)` begins with the generation of the basic ontology. This is done by the `genOntology()` function. In fact, this function generates an ontology (file cve.owl), which includes (import) the ontologies of the given years, i.e. does not include ontologies of all years.

The results are saved in the subdirectory “results”.

If the download parameter for a fresh copy of the database from the site is set, then the `downloadNVD()` function is called. The last function obligatorily downloads the files nvdcve-1.1-modified.json.zip and nvdcpematch-1.0.json.zip, but for the years it first downloads the respective meta-file and checks by date in it whether the copy that is available differs from that on the site (the available files are possibly older), if there is a difference then download the corresponding file. If a file is not downloaded at all, then it is downloaded. The checks are made by the function `is_downloaded(name)`, and the download itself from the `download(url, fileName)`.

There is a danger here that if the system crashes during the download and the meta-file may have been downloaded but the file itself is not. In this situation, it is recommended simply to clear the entire contents of the "data" directory and then run the program again with the desired parameters.

As noted above, the cve.owl file contains an ontology that includes the selected generation ontologies. In the file nvd.owl the description of the ontology (shell) is generated – this is the next step in the execution of the program.

The next step is to create a dictionary of configuration-base CPE names. The contents are found in the file nvdcpematch-1.0.json. A secondary structure of the scheme is used, due to which the control is weak. Fields that do not belong to them or empty configurations are published in this file from time to time, i.e. without base names. The latter are saved in the log file of the respective part (log extension).

However, this dictionary is central to the generation of configurations.

The further processing continues in parallel mode by years.

The file with the changed vulnerabilities is always processed first. For this purpose, the standard process is run with appropriate parameters `processFeed(inQueue, resultQueue, cpeFeeds, modified = None, onlyModifiedCVEs = False)`. There are two options for this processing: to include the modified vulnerabilities in the resulting ontology or not. In any case, it is necessary to know which are changed vulnerabilities are so that they are not re-included in the ontologies of the respective years.

The processing of the modified vulnerabilities is done in parallel with the main process, although this is unnecessary, but the scheme of work this process is more convenient to be slightly modified for the case than other code to write. The processing process itself is described below.

After processing the modified vulnerabilities (or obtaining at least the set of their names), the processing itself begins.

First, as many parallel processes as there are cores in the machine are created. In the `inQueue` input queue for them, the main process loads the names of the years for which ontologies will be generated. If is planned to be generated for an ontology for the modified vulnerabilities, it is skipped, as this has already happened in the previous step.

When the main process completes loading the input queue, it sends one `DONE` "signal" for each of the processes. This is a message loaded in the queue. It is a string and the process stops when it reads that string.

The loading of the input queue takes place in parallel with the operation of the processes. After loading the queue, the process `main(download, feeds)` waits for the processes to finish and then ceases to exist.

Processes communicate with the main process through the `resultQueue` queue. By default, it sends the number of processed vulnerabilities and an end signal. When modified vulnerabilities are processed, the set with their names is sent to this queue. The latter, by default, is transmitted during the initialization of the main parallel processes.

The parallelism scheme is shown in Figure 1.

Parallel processes process one piece (year) of vulnerabilities. In fact, the level of parallelism is by years.

The `processFeed()` process reads the name of another year from the `inQueue` queue. Stops (exits the cycle) when it receives a `DONE` signal.

It then reads the file of its year and begins to cycle through its vulnerabilities.

In order to be able to read the year in Protégé, the year is presented as a separate ontology, but it is also very large and is therefore broken into parts. Each part can be used independently.

In addition, the configurations (CPEs) to which the vulnerabilities are associated are very large and are therefore exported to a separate file, which is also broken down into parts as separate ontologies due to their size.

In fact, each part of the year has its own series of configurations, which allows both the part and the ontology of a separate configuration to be loaded independently.

There are two template files for this purpose: shellPart.owl and shellConfig.owl, which are used to create ontologies.

The mechanism of part generation and automatic switching from file to file is implemented with the `BF` class, which in its methods resembles an ordinary file, but it takes care of breaking the year into parts and configurations of a part into individual subparts.

The `BF` class contains two variables, `ontoPart` and `ontoConfig`, which are initialized with the contents of the shellPart.owl and shellConfig.owl files. These are the shells (templates) for part ontology (year, feed) and CPE configurations.

The `newPart()` and `newCPart()` methods create the next file from the corresponding part. Templates are also used there. The logic of these methods is related to the file organization of bait ontologies. For example, for the year 2017 a file o2017.owl is created, which contains only the inclusion of the ontologies generated for the year. In this case, these are ontologies in the o20170.owl to o20178.owl files.

The names of the ontologies are after the names of the files, i.e. without the "owl" extension.

A file with a number added to the name of the main ontology is created for each subpart. Each subpart is a separate ontology and can be used as such. These ontologies contain descriptions of the vulnerabilities only, but not of the platform configurations (CPE Names) they refer.

Platform configurations are found in a series of files that begin with the file name of the subpart, but with an added number preceded by a "c". For example, the o20170.owl partition has o20170c0.owl configuration files up to o20170c3. owl. These files are also stand-alone ontologies that have names in their files, similar to the subdivisions of the part.

The `BF` class supports this organization files.

The `BF` writing methods are `write()` and `cwrite()`. The first is for writing in the file of the part (`file`), and the second – in the configuration file (`cfile`).

The size of the files (subpart and configurations) is controlled by the `flush()` and `cflush()` methods. If the maximum limit is exceeded, then the current subpart or configuration is closed and new files are created.

Closing both files is done using the `close()` and `cclose()` methods.

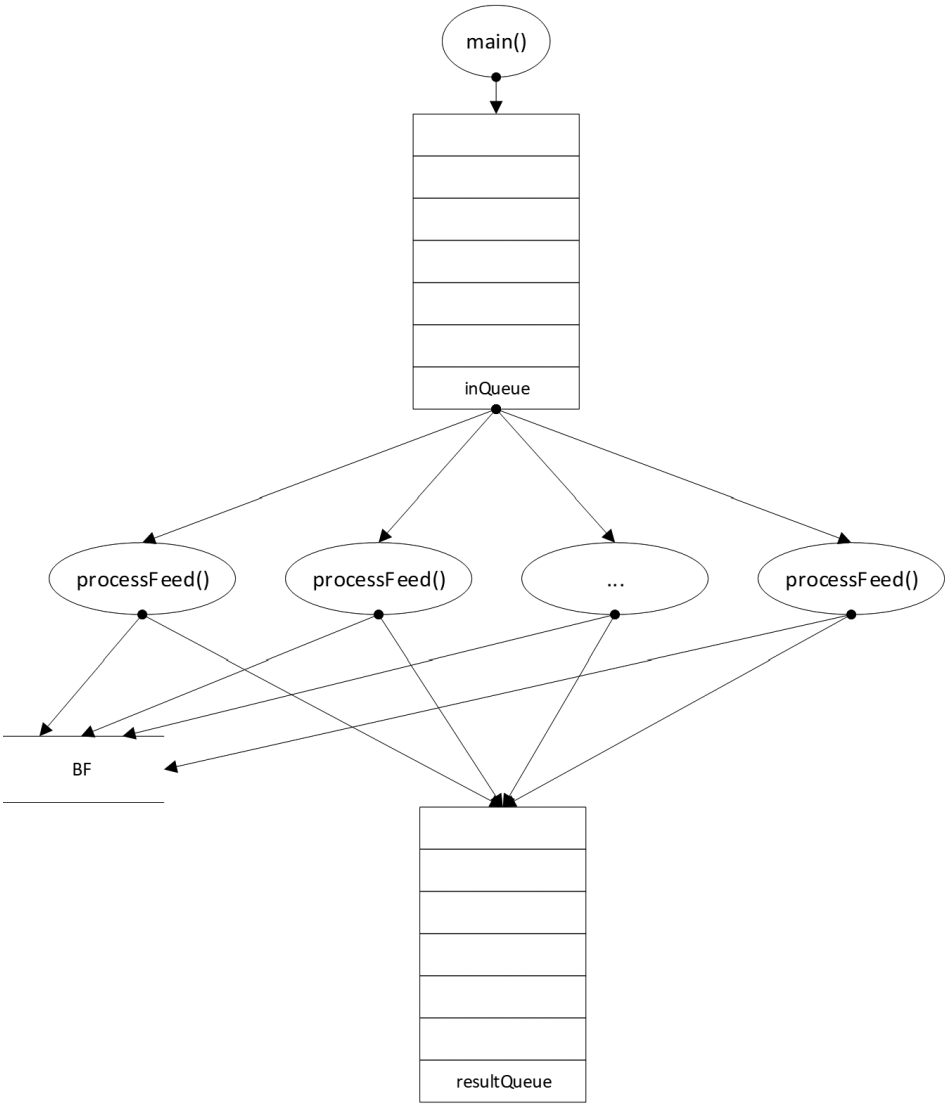

Figure 1: Scheme of the parallelism

There are two other help methods, `getOntoName()`, which returns the name of the current ontology, and the `parts()` method, which returns the number of subparts generated.

The very work of the process `processFeed(inQueue, resultQueue, cpeFeeds, modified = None, onlyModifiedCVEs = False)` consists in traversing the structure of the individual vulnerability and the generation of the respective individuals from this description.

Because `processFeed()` works as an isolated process, some basic means of communication need to be passed as parameters. This is the `inQueue` input queue that receives the baits and the results queue that sends the results. As a result, it returns the number of individuals generated. When working on the Modified, the `result` queue also returns the set of names of modified vulnerabilities. When a set of names of the modified parameter is given, it is understood that it is working on a standard bait, otherwise this parameter is `None` by default and means that it is working on Modified, i.e. many of the names of the modified vulnerabilities will be generated. However, if the `onlyModifiedCVEs` parameter is true, then an ontology of the modified vulnerabilities will be generated, but this is not done by default because the parameter is false.

The `cpeFeeds` parameter is a dictionary that has all keys from `def_cpe_match` to nvdcpematch-1.0.json without cpe_name. The key thus formed uniquely identifies an element of compliance. The value in the dictionary elements is a list of CPE names – basic, not templates.

The internal function `genFeedOntology(feedName, parts)` of the process function generates a description of the bait ontology.

The generation of vulnerabilities is in the body of `processFeed()`, and in a separate internal function `generateProps(item)` the generation of configurations is outlined.

The body of `generateProps()` generates descriptions of both versions of the CVSS metric and some other simple vulnerability properties. The internal function `writeFacts(txt)` is widely used here, as it is code with a high frequency of use and otherwise obscures the readability of the code.

In a separate internal function of `generateProps()` is a separate function for generating effects `generateAffects()`. This information is obviously outdated and is from older versions – missing in the NIST dictionary, but left as a description in the diagram. Eventually, it can be removed in the future.

The configuration text is generated from `generateConfigurations(outFile)` to `generateProps()`. This is a relatively simple function, but relatively difficult to understand and work with is the other internal `findNode(node)` function, which generates CPE configurations, and its internal `findMatch(match)` function. The latter two functions have been developed in the CPE language for applicability of compliance [2]. The reason for this is that there is no clear description provided by NIST of the application of the language in this case. This of course leads to a number of problems in the dictionary itself for the generated configurations. For example, CPE names are missing. Maybe in the future the situation will change. However during the development of the generator for a relatively short period of 2-3 months were generated by NIST files for configurations that were incompatible and these necessitated refinements to avoid these problems. De-

tailed diagnostics for missing CPE names are also provided. The weak control between the applicability language expressions and the configurations in the CPE dictionary is obvious.

The first `findNode(node)` function works with nodes in the applicability language. A node is an operation with its operands.

The second `findMatch(match)` function works on CPE templates. Here we are talking about templates in the language of applicability, but the problem is that the CPE names themselves can be considered in particular as templates.

Finally, there is another common function `codeString(s)`. It deals with the annoying, but intense, encryption of the Python escape symbol in OWL.

Due to the high level of parallelism, the application of functions is applied here in order to avoid possible errors in copying the environment in the individual processes. In this course of the strategy, the `cpeFeeds` configuration dictionary is also copied.

3. Parallelism formalization

The parallelism used here is at the level of feeds. As explained above, the Modified feed is first processed in a separate process, as it contains the changes made to the feed of previous years. If Modified is included in the list of processed feeds, then its ontology is generated, but at least the set of names of modified vulnerabilities that are not processed in other feeds, i.e. for them no individuals are generated in the respective ontologies.

The same function is used to generate the ontologies of the individual feeds and the Modified ones, as there is too much common code and small differences. In the specification of parallelism, however, these two behaviors are presented as separate processes for greater clarity.

The function, which implements the process of generating ontologies for the individual feed, is performed sequentially and therefore the details of its work are reduced to the level of parallel interaction. In fact, this function generates a general ontology for the entire feed by including its individual parts, which can also be opened independently as ontologies in Protégé. For all feeds, a common ontology is also generated by including the feeds so that they can be opened together as one ontology.

This approach of building independent ontologies through inclusion was chosen because of the huge ontology volume. However, and in order to be able to control the content of the individual parts by opening them in Protégé.

The CSP specification code [6] in PAT version 3 [7] is:

```
#define queueBufferLength 10;
#define noProc 4;
channel inQueue queueBufferLength;
```

```
channel resultQueue queueBufferLength;
channel sync[noProc] 1;
channel msync 1;
#define mModified 1;
#define mSet 2;
#define mDONE 3;
#define mFEED 4;
#define mS 5;
#define mCount 6;

processFeedModified() =
   (
          inQueue?mModified -> produceModifiedFeedOntology ->
          resultQueue!mS -> resultQueue!mCount ->
          resultQueue!mDONE -> Skip
          [*]
          inQueue?mSet -> produceModifiedSet -> resultQueue!mS ->
          resultQueue!mDONE -> Skip
   ) ;
   inQueue?mDONE -> resultQueue!mDONE -> msync!mDONE -> Skip;
processFeed(i) =
          inQueue?mFEED -> produceFeedOntology ->
          resultQueue!mCount -> processFeed(i)
          [*]
          inQueue?mDONE -> resultQueue!mDONE -> sync[i]!mDONE ->
          Skip;
main(first) =
   if (first) {
          (
             processFeedModified() |||
             (
                 MODIFIED -> inQueue!mModified ->
                 inQueue!mDONE -> resultQueue?mS ->
                 resultQueue?mCount -> resultQueue?mDONE -> Skip
                 [*]
                 SET -> inQueue!mSet -> inQueue!mDONE ->
                 resultQueue?mS -> resultQueue?mDONE -> Skip
             )) ;
          resultQueue?mDONE -> msync?mDONE -> (main(false) |||
             (|||i:{0..noProc-1}@processFeed(i)))}
   else {
          FEED -> inQueue!mFEED -> resultQueue?mCount ->
          main(false)
          [*]
          END -> (|||{noProc}@inQueue!mDONE -> Skip) ;
```

```
                    (|||{noProc}@resultQueue?mDONE -> Skip) ;
                    (|||i:{0..noProc-1}@sync[i]?mDONE -> Skip)};
System() = main(true);
#assert System() deadlockfree;
#assert System() divergencefree;
#assert System() deterministic;
#assert System() nonterminating;
```

The `queueBufferLength` buffer size definition has been reduced to 10, which is not the case in Python – there the queues can be considered infinite, although in reality they have a limit.

The number of `noProc` processes is modeled by the number of cores in the test machine – 4.

The `inQueue` and `resultQueue` queues are modeled by channels.

The synchronization to complete the `sync[noProc]` workflows, the Modified processing `process(msync)`, and the main process is modeled with buffered channels 1.

The messages exchanged on the channels are numerically modeled as follows `mModified(1), mSet(2), mDONE(3), mFEED(4), mDict(5)` and `mCount(6)`. In the model, the message values are not important for the parallelism, only their type.

The system is modeled through the `System()` process, which is very similar to the implementation.

The main (first) process is divided into two parts: Modified bait processing and standard bait processing.

In the first case, `processFeedModified()` is started in parallel with overlap `processFeedModified()`. There are two ways to do this: when Modified is included in the ontology (`MODIFIED` event) and when it is not (`SET` event). In the first case, the `mModified` and `mDONE` messages are loaded in the `inQueue` queue to inform the process that only the Modified will be processed, but with ontology generation. The `processFeedModified()` process returns the `resultQueue` set of modified vulnerability names via the `mS` message and the number of individuals generated in the ontology `mCount`. The same queue informs that the work with the `mDONE` message is completed.

Similarly, the process operates in the second case, but then does not return the number of generated individuals.

Both cases continue with the completion of the `processFeedModified()` process on both `resultQueue` and `csync` queues, i.e. receive a `mDONE` message on each of them.

The `mDONE` message is used to inform that the process itself has completed its processing operation and to inform it that it is shutting down. The second type of information is disseminated in both queues, as there is a difference in the fact

that the process has completed its work, but the recording of the results has not been completed. For this reason, although not very clear from the specification, the mDONE message is transmitted a second time in the resultQueue queue.

Finally, the first start of main (first) ends with the parallel start of workflows and main (false), i.e. the second part of the main process.

In fact, as many processFeed(i) processes are run in the implementation as there are feeds if they are less than the number of CPU cores, or if there are more, as many as the CPU cores. In this case, it is firmly modeled with 4 processes as many as the cores of the test machine, but this is not a problem from the point of view of parallelism.

The second part of the main process, false, processes the individual baits (FEED event) and terminates the system (END event).

The processing of the baits consists in loading in the queue inQueue a message for the next bait mFEED and then reading from the queue the results of the message for the number of generated individuals, after which there is recursion in the second part of the main process, i.e. address main(false).

In an END event, i.e. the processing are exhausted, as many mDONE end messages are loaded in the input queue, as there are workflows. The same number of messages is then read from the results queue. Finally, the sync[i] sync channels read the same end-of-work message. In the specification, these steps are presented in parallel, but in the implementation, the steps are implemented sequentially.

The processFeedModified() process begins with two alternatives to the contents of the input queue. When the mModified message is received, it bears the name of the bait, processes it (produceModifiedFeedOntology event), displays a set of modified vulnerability names with the mS message in the resultQueue queue, the number of individuals generated in the ontology, the end message (again in the same queue), and finally completes the implementation of this branch.

The second alternative (mSet event) is analogous to the first, with no return of the number of generated individuals, and the processing is for the produceModifiedSet event.

After processing the alternatives, the processFeedModified() process waits for an end message mDONE, then displays the end message in the resulting queue and in its csync sync channel.

In the implementation, the processFeedModified() process is a modified processFeed(i) process with direct control from the main process, but it is also parallel. The idea is one code for both processes.

Here in the specification processFeedModified() waits for an end message after executing one of the two alternatives, but in fact this logic is set by the main process in the implementation, and in the specification.

The `processFeed(i)` workflows handle two events: the next feed (`mFEED` message) and the end of work (`mDONE` message).

The generation of the ontology is modeled with the `produceFeedOntology` event. The number of generated individuals is then displayed in the resulting queue and a recursive reference to the same process follows.

The order of the number of generated individuals does not matter, because in the main process it is summed up for the whole ontology, not for the individual parts. In fact, this amount is informative and does not participate in ontologies.

Processing events (such as `produceFeedOntology`) in both workflows `processFeedModified()` and `processFeed(i)` are intrinsic in nature and have nothing to do with process synchronization (i.e., overlapping concurrency), but are complex to illustrate the logic of work.

The second processing that the `processFeed(i)` process does is end (`mDONE`). The process then displays the `mDONE` end message in the resulting queue and in its `sync[i]` sync channel, and `Skip` successfully completes the job.

Finally, the specification contains the standard checks for the system, which pass successfully when verifying the expected behavior.

4. Conclusion

The generated ontology of vulnerabilities exceeds terabytes in volume. The management of such a huge ontology is possible only by breaking the parts and therefore when starting the generator it is specified which years of vulnerabilities to be generated.

It is possible to generate and publish the entire ontology on the Web, but the server resources must be large in order for the entire ontology to be used and shared.

Unlike the CPE ontology, the NVD ontology is too large to be manipulated in Protégé as a single file.

The parallelism in the generator is fixed by the number of cores and is at the level of the year, i.e. if released in just one year, there will be virtually no parallelism.

Breaking parallelism at a lower level is not justified.

With the CPE ontology generator, each platform can be processed independently and the generation result can be inserted into one ontology file. The ordinance of the individuals in this file does not matter.

The NVD ontology generator requires that all individuals in a part be linked to the ontologies of their configurations, i.e. generation is in order: generating the individual vulnerability followed by generating vulnerability configurations.

In principle, the level of parallelism could be raised by generating configurations in parallel. This can be achieved by redesigning the ontology: the individual parts of the configurations are combined into one ontology of configurations, which is included in each part of the ontology of the year. With this approach, the size of the configuration ontology will be very large and will most likely not be able to load into Protégé.

However, there is another problem here, the number of cores is limited, and i.e. the number of truly parallel processes is limited. No matter how much we increase parallelism, it depends on the number of cores.

A distributed parallel version of several computers is also possible. This is quite easy to do with Python, but it seems the most reasonable is the chosen level of parallelism in this case as well.

In the case of the distributed variant, there is also the issue of collecting the results of the generation. The data is huge; it is possible to be redirected to a server with significant disk space. This functionality is easy to implement, but network traffic will jump sharply.

In the distributed version, the data may remain on the computers on which it is generated – something like the Map phase in MapReduce computing [4], but there must be greater clarity about their use, i.e. whether there will be a Reduce phase or directly where the ontologies are located there will be accessible. This is a mixture of MapReduce and access to the Semantic Web.

Let us return to the parallel variant by separating the generation of individuals from that of configurations. Another problem is that the mechanisms of data transmission are quite heavy. It is worth transferring some data to the process and it will take much longer, and most likely, this will not be the case, i.e. there will be a lot of heavy traffic that will exceed the calculation process. Something similar to the problem in CORBA [5] where the traffic between sites significantly exceeds the calculation process in them.

Of course, the considerations given here are of the most general nature and considerable research effort is needed to reject or validate the hypotheses raised here about parallelism and distribution on several computers.

5. Acknowledgements

This paper is prepared with the support of MIRACle: Mechatronics, Innovation, Robotics, Automation, Clean technologies – Establishment and development of a Center for Competence in Mechatronics and Clean Technologies – Laboratory Intelligent Urban Environment, funded by the Operational Program Science and Education for smart growth 2014–2020, Project BG 05M2OP001-1.002-0011.

6. References

[1] NIST, Information Technology Laboratory, National Vulnerability Database (NVD), 2022. URL: https://nvd.nist.gov

[2] NIST, NISTIR 7698, Common Platform Enumeration: Applicability Language Specification Version 2.3, 2022. URL: https://csrc.nist.gov/publications/detail/nistir/7698/final

[3] Musen, M.A. The Protégé project: A look back and a look forward. AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, 1(4), June 2015. DOI: 10.1145/2557001.25757003.

[4] Dean, J. and S. Ghemawat, MapReduce: Simplified data processing on large clusters. In Proceedings of Operating Systems Design and Implementation (OSDI). San Francisco, CA. 137-150, 2004.

[5] CORBA. URL: https://www.corba.org

[6] Hoare C. A. R., Communicating Sequential Processes, 1985–2004