Decision support system for decomposing functional software between plugins

Alexey D. Shabliy

Moscow Aviation Institute  
Moscow, Russia  
alexey.shabliy@gmail.com

Second Author

Group, Laboratory, Address,  
City, State ZIP/Zone, Country  
secondauthor\_id@domain\_name

Received (Day Month Year)  
Revised (Day Month Year)  
Accepted (Day Month Year)  
Published (Day Month Year)

This article discusses the construction and application of a mathematical model for a decision support system for decomposing software functionality between plugins. The constructed graph model is used to solve an optimization problem, the initial data for which are formed based on the results of analyzing a real open source project. A comparison of the results of solving the optimization problem using the reinforcement learning method and a genetic algorithm is performed. A conclusion is made about the prospects for further research in order to complicate the objective function to implement the possibility of optimizing not the volume of functionality, but the cost costs of software.

*Keywords*: Software, plugin, decomposition, graph, reinforcement learning, genetic algorithm.

2020 Mathematics Subject Classification: 05C90, 90C35

# Introduction

Currently, software decomposition is a pressing issue in the field of information technology. This can be partly explained by the growing popularity of distributed systems, which have become the flagship direction in this field in recent years and for which functional decomposition is a fundamental task.

Microservice architecture is a leading solution in building distributed systems. And it is no exception in the matter of the importance of functional decomposition. Thus, in the Refs. 1–3 a description of the principles and patterns is given, guided by which the functional decomposition between microservices should be performed.

As an alternative, the Ref. 4 provides an example of using artificial intelligence technologies for clustering a subject area with the aim of further dividing the clusters into microservices. The efficiency is confirmed by objective costs for two indicators: message-level connectivity and domain-level connectivity.

However, decomposition can be performed not only from the aspect of the complexity of the subject area, but also based on the number of available hardware capacities on which future microservices will have to perform their work. In Refs. 5–7, the performance aspect is considered and compared with the indicators of a solution on a monolithic architecture. The comparison criteria are the application runtime, labor costs for deployment and maintenance.

In client-server solutions, microservice architecture is the leader in the server part of the application. As its analogue in the client part, we can highlight plugin systems. One of the most popular solutions that uses plugin system technology is WordPress (see Ref. 8), which is used by millions of sites and has almost 55,000 (fifty-five thousand) available plugins. However, plugins, performing their work in a single information environment presented in the form of a web page, can conflict with each other and generate anomalies during operation. One such example is the processing of html tags. WordPress plugins supplement the html page with various tags, and the executable JavaScript code does not identify the difference, that the tag could have been added in an undeclared format, which leads to conflicts and anomalies (see Refs. 9, 10).

Another example of a problem is the delivery of software under different licenses (see Refs. 11, 12). Depending on the purpose and distribution capabilities, the software must include or not include components that involve dependencies with certain distribution restrictions.

The problems outlined can be solved by decomposing the functionality and delivering it not as a single plug-in, but as a complex. Decomposing the functionality makes it possible to deliver software in various packages, the feasibility of which is determined by the customer. By using packages with a limited amount of functionality, the customer can ensure that code that leads to an abnormal situation or problems with licenses does not get into their system.

The need to perform decomposition, as well as the complexity of its implementation, justify the need to create a decision support system (DSS) for decomposing software functionality between plugins. The initial data for its operation should be:

* the software to be decomposed,
* number of plugins to be decomposed into functionality,
* formula for calculating the objective assessment of the completed decomposition.

In the current study, the completed decomposition is assessed by the number of functional software requirements implemented in the package: the closer this value is to the number of useful requirements in the package, the fewer useless ones are implemented in the package.

The purpose of this work is to describe the mathematical model of the DSS and demonstrate its operation when interacting with the code base of a real project. The work consists of dividing the code base between plugins in accordance with the declared sets of useful functional requirements.

The objectives of the work are:

* description of the entities of the subject area and the nature of their interaction,
* creation of a mathematical model,
* analysis of a real project and calculation of the initial data values ​​for the operation of the compiled mathematical model,
* conducting computational experiments and assessing the performance of the model.

# Model Description

## Graph

The entities of the subject area are:

* functional requirements - they describe the functionality of the software,
* source code files - they implement functional requirements in a programming language,
* plugins - software integration units into a plug-in system, a plug-in included in the delivery.

Entities are characterized by:

* a requirement is considered implemented if all source code files implementing it are included in the package,
* for a file to be included in a package, the plugin containing it must be included in the package,
* if a plugin is included in the package, it includes all the files it contains.

Entities enter into the following interactions:

* one file can be used to implement multiple requirements,
* one requirement can be implemented in multiple files,
* source code files have dependencies between each other and all dependencies must be resolved to enable delivery,
* files are distributed among plugins, and one plugin can include several files, but one file cannot be located in several plugins at the same time.

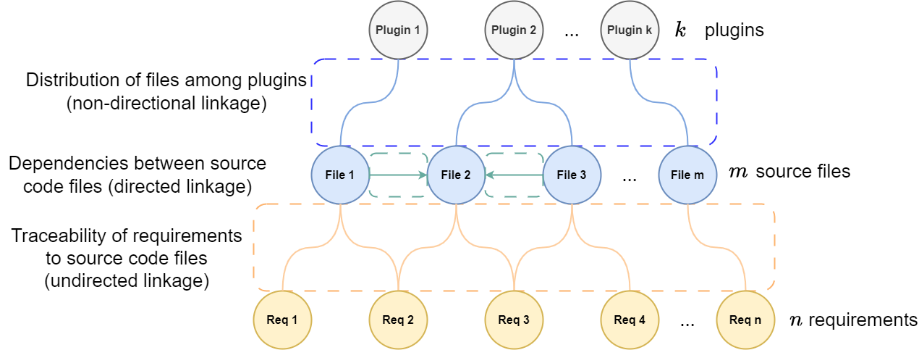
In this way a graph is formed in which the vertices are indexed and constitute:

* functional requirements,
* source code files,
* plugins.

Edges in a graph are formed when:

* tracing requirements to source code files - links,
* the presence of dependencies between files - links,
* distribution of files between plugins - links.

The given description corresponds to the graphical mathematical model shown in the figure 1.

Fig. 1.  Graph mathematical model.

The described graph model, using information about the designated vertices and edges, allows us to determine the composition of connectors for delivery (algorithm 1) and the requirements to be implemented (algorithm 2) in each of the declared configurations.

### Algorithm 1

For algorithm 1 the initial data are:

* composition of useful requirements within the framework of the set,
* traceability of requirements to source code files,
* dependencies between source code files,
* distribution of source code files among plugins.

Execution of the algorithm 1:

* identifying source code files that implement useful requirements,
* resolving dependencies between files,
* defining the composition of plugins, which includes all the files involved.

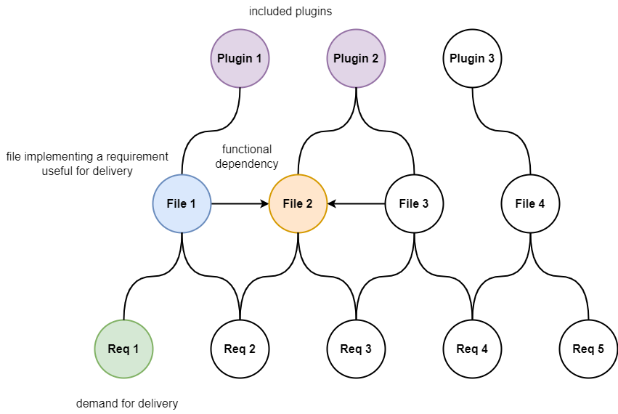
An example of executing algorithm 1 is shown in the figure 2:

Fig. 2.  Example of execution of algorithm 1.

### Algorithm 2

For algorithm 2 the initial data are:

* composition of plugins included in the package,
* distribution of source code files among plugins,
* traceability of requirements to source code files.

Execution of the algorithm 2:

* identifying all files contained in plugins,
* definition of requirements for the implementation of which all necessary source code files are supplied.

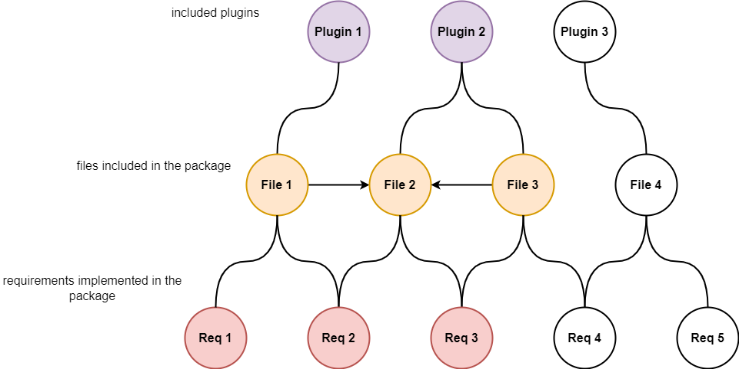
An example of the execution of algorithm 2 is shown in the figure 3:

Fig. 3.  Example of execution of algorithm 2.

Conclusion: with a given distribution of files by plugins and the required one requirement, three requirements are implemented within the package.

## Analytical formula

To construct an analytical formula, it is necessary to describe the vertices and edges of the graph using mathematical objects and restrictions imposed on their values.

### Traceability

Traceability of requirements to source code files can be conveniently described as a matrix , the elements of which take values ​​in the range . In it, the row number corresponds to the requirement index, and the column number to the file index.

The elements of take the values:

* if the -th file is involved in implementing the -th requirement,
* if the -th file is not involved in implementing the -th requirement.

It is assumed that all requirements are met, therefore the condition is introduced.

### Dependencies

Dependencies between source code files are conveniently described as a square matrix of binary relations . In it, the line and column numbers correspond to the file indices. The elements of take the values:

* if the -th file is dependent on the -th,
* if the -th file is independent of the -th.

The file is considered independent of itself, so .

### Distribution

The distribution of source code files between plugins can be conveniently described as a matrix of binary relations . In it, the row number corresponds to the file index, and the column number to the plugin index. The elements of take the values:

* if the -file belongs to the -plugin,
* if the -file does not belong to the -plugin.

It is assumed that all files are distributed among plugins, so the condition is introduced.

### Equipment

Within each of the configurations, useful requirements can be conveniently described as a vector of binary relations. For configurations, such vectors form a matrix of binary relations . In it, the row number corresponds to the configuration number, and the column number to the requirement index. The elements of take the values:

* if within the -th configuration the -requirement is useful,
* if within the -th configuration the -requirement is useless.

### Useful source code files

Files implementing useful requirements in each of the $l$ configurations can be conveniently described as a matrix . In it, the row number corresponds to the configuration number, and the column number to the file index. The values ​​of the elements of the matrix are:

. (1)

Resolving dependencies of matrix files:

. (2)



Resolving dependencies of matrix files:

. (3)

It is obvious that:

. (4)

Note that the dependency resolution depth cannot exceed the value , which means the total number of files that must be included in the delivery is:

. (5)

### Plugins

Plugins that form a delivery in each of the configurations can be conveniently represented as a matrix . In it, the row number corresponds to the configuration number, and the column number to the plugin index. The values ​​of the elements of the matrix are:

. (6)

The function is auxiliary and is used to transform the values ​​of matrix elements:

. (7)

The obtained values ​​of the matrix should be interpreted as follows:

* within the -th configuration, the -th plugin must be installed if ,
* within the -th configuration, the -th plugin should not be installed if .

### Supplied files

The files included in each of the packages can be conveniently represented as a matrix . In it, the row number corresponds to the file index, and the column number to the package number. The values ​​of the elements of the matrix are:

. (8)

### Useful requirements

The requirements implemented in each of the $l$ configurations can be conveniently represented as a matrix . In it, the row number corresponds to the requirement index, and the column number corresponds to the configuration number. The values ​​of the elements of the matrix are:

. (9)

The function is auxiliary and is used to transform the values ​​of matrix elements:

. (10)

The obtained values ​​of the matrix should be interpreted as follows:

* within the -th configuration, the -th requirement is implemented if ,
* within the -th configuration, the -th requirement is not implemented if .

In this way, an optimization problem with a minimization objective function can be formulated and solved:

. (11)

# Initial Data

It was assumed that the initial data would be based on open source software that would be useful for more than one category of users in terms of its functionality, and would also provide a range of functional capabilities for the purposeful division of it into different configurations.

The open source web application “meta-configurator” (see Refs. 13–16) was chosen for the experiments. It generates a graphical user interface depending on the scheme in which the user has loaded the document. This allows for a significant reduction in the costs of developing and supporting a specific interface for a particular format.

It uses a schema-to-user interface approach with three key features:

* provides a unified presentation that combines the advantages of both a graphical interface and a text editor,
* provides a schematic editor,
* supports advanced schema features including conditions and constraints.

The results of the research conducted by the developers among users indicate the effectiveness of the approach they proposed for extracting information from data and diagrams, as well as editing them.

In the context of this work, working with users, as well as a wide range of software functionality, determines the relevance of applying the theoretical results of the study to this practical example. For example, it would be possible to separate and deliver independently work with schemes in JSON and XML formats or data management and scheme management.

An analysis of the software code base (<https://github.com/MetaConfigurator/meta-configurator.git> at revision 6068f048) was carried out in order to identify:

* functional requirements,
* source code files,
* traceability of requirements to source code files,
* dependencies between source code files.

Also, based on the results of the analysis, the proposed compositions of the equipment should have been formed.

Table 1.  Volume of initial data.

|  |  |  |
| --- | --- | --- |
| Language (format) | Number of files | Number of lines |
| TypeScript | 143 | 19095 |
| JSON | 12 | 12397 |
| Vuejs Component | 56 | 6656 |
| Text | 13 | 195 |
| JavaScript | 4 | 66 |
| Markdown | 2 | 68 |
| YAML | 2 | 28 |
| Dockerfile | 1 | 22 |
| HTML | 1 | 15 |
| CSS | 1 | 5 |
| Total | 241 | 38547 |

Not all analyzed files contain source code. In addition, not all source code files are used to implement any functional requirements. The analysis showed that some files contain debugging procedures, are configuration files, or describe project build scenarios. The following was generated based on the analysis results:

* a list of functional requirements that a software tool implements (<https://github.com/AlexeyShabliy/codebase-overview/blob/main/requirements.csv>),
* a list of source code files that implement the identified functional requirements (<https://github.com/AlexeyShabliy/codebase-overview/blob/main/files.csv>),
* traceability of functional requirements to source code files (<https://github.com/AlexeyShabliy/codebase-overview/blob/main/tracer.json>),
* dependencies between source code files (<https://github.com/AlexeyShabliy/codebase-overview/blob/main/dependencies.json>),
* list of configurations indicating the composition of the requirements implemented in them (<https://github.com/AlexeyShabliy/codebase-overview/blob/main/complectations.json>).

The results of the analysis are converted into matrices , ,, having the following dimensions:

* ,
* ,
* .

# Experiments

Computational experiments were conducted to confirm or refute the hypothesis that with an increase in the number of plugins, the amount of useless functionality in relation to all possible configurations will decrease. To carry out the calculations, equipment with the following characteristics was used:

* operating system Ubuntu 23.04,
* 2-core Intel Core i5 processor with a clock speed of 1.8 GHz,
* RAM capacity 8 GB.

The computational experiments consisted of finding the optimal values ​​of the matrix that would maximize the inverse value of the objective function. Two modes of operation of the algorithm for searching for values ​​of the matrix were considered:

* - vector of length consisting of binary values,
* - vector of length consisting integer values ​​in the range .

Convert to :

. (12)

Convert to :

. (13)

Analysis of the variants of the solution generated by the algorithm showed that the transformation of into guarantees the fulfillment of the condition , in addition, the length of the vector is less than the length of , which simplifies the search for the optimal solution. Therefore, it was decided to generate the vector at each iteration of the algorithm.

Reinforcement learning (RL) is one of the leading methods based on artificial intelligence technology (see Ref. 17). The ability of RL to perform continuous learning determines the feasibility of its use for solving the formulated optimization problem, and the use of a genetic algorithm is considered as an analogue of RL (see Refs. 18, 19).

The configuration of the applied RL is given in table 2.

Table 2.  RL configuration.

|  |  |
| --- | --- |
| Configuration parameter | Number of lines |
| Number of iterations | 1000 |
| Space of action | MultiDiscrete |
| Observation space | Box |
| Number of dimensions of observation space | *k* |
| Algorithm | A2C (see Ref. 20) |

The configuration of the applied genetic algorithm is given in table 3.

Table 3.  Genetic algorithm configuration.

|  |  |
| --- | --- |
| Configuration parameter | Number of lines |
| Number of generations | 1000 |
| Number of chromosomes | 4 |
| Number of genes | *m* |
| Crossover type | single-point |
| Type of mutation | swap |
| Percentage of mutated genes | 10 |

A comparison of their performance was performed on a problem of lower dimension, in which , . The results of the performance comparison are given in table 4.

Table 4.  Comparison of algorithm performance.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | genetic result | RL result | genetic duration, sec | RL duration, sec |
| 1 | 161 | 161 | 1.2 | 52.5 |
| 2 | 161 | 161 | 1.7 | 57.0 |
| 3 | 161 | 161 | 1.7 | 57.0 |
| 4 | 161 | 161 | 1.7 | 56.8 |
| 5 | 161 | 161 | 1.6 | 56.8 |
| 6 | 161 | 161 | 1.8 | 57.2 |
| 7 | 161 | 161 | 1.9 | 57.2 |
| 8 | 161 | 161 | 1.8 | 57.7 |
| 9 | 161 | 145 | 2.0 | 56.8 |
| 10 | 159 | 142 | 1.7 | 56.6 |

Performance evaluation showed that in order to increase the efficiency of the genetic algorithm, it is possible to run it multiple times with subsequent analysis of not one, but many results. For example, the minimum, maximum, and average values ​​can be taken into account. This is unacceptable for RL, since it requires significant time expenditures for calculations.

A series of computational experiments were conducted to solve the optimization problem for different values ​​of the coefficient in the range . Computational experiments using the genetic algorithm for each of the values ​​of were conducted 100 (one hundred) times with subsequent determination of the maximum, minimum and average values ​​of the objective function. For RL, the computational experiment for each of the values ​​of was performed once.

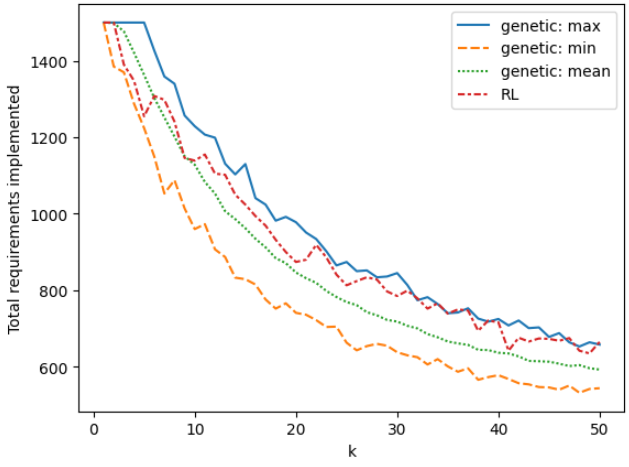
The figure 4 shows a graphical interpretation of the computational experiments performed.

Fig. 4.  Experimental results.

The results of the experiments show that for the software project under consideration, the use of the DSS mechanism developed in this work is relevant. This is confirmed by the decrease in the total number of implemented requirements in the supplied software packages with an increase in the number of plug-ins, by which the decomposition of functionality is performed.

# Conclusion

The article proposes a mathematical model for DSS. It has been tested on a real open source project using various optimization algorithms, including those using artificial intelligence technology.

In further studies, it is proposed to refine the objective function so that it takes into account not the requirement implementation indicator, but the amount of costs for software development or support. In addition, it is proposed to perform a mathematical formalization of the optimization problem in order to use software solvers for its solution, such as GNU Linear Programming Kit and COIN-OR Branch-and-Cut, as well as subsequent comparison with the methods for solving the optimization problem presented in this paper.

# Acknowledgments

?

# ORCID

# Alexey D. Shabliy - <https://orcid.org/0009-0003-9632-0251>

# References

1. Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, P. Clarke, Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review, *IEEE transactions on software engineering*, Vol. 49, No. 8, August 2023, pp. 4213–4242, <https://doi.org/10.1109/TSE.2023.3287297>.
2. C. K. Rudrabhatla, Impacts of Decomposition Techniques on Performance and Latency of Microservices, *(IJACSA) International Journal of Advanced Computer Science and Applications*, Vol. 11, No. 8, 2020, <https://doi.org/10.14569/IJACSA.2020.0110803>.
3. V. Velepucha, P. Flores, A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges, *IEEE Access*, ISSN 2169-3536, Vol. 11, August 15, 2023, pp. 88339–88358, <https://doi.org/10.1109/ACCESS.2023.3305687>.
4. O. Al-Debagy, P. Martinek, A Microservice Decomposition Method Through Using Distributed Representation of Source Code, *Scalable Computing: Practice and Experience*, ISSN 1895-1767, Vol. 22, Issues 1, February 9, 2021, pp. 39–52, <https://doi.org/10.12694/scpe.v22i1.1836>.
5. M. H. Hasan, M. H. Osman, N. I. Admodisastro, M. S. Muhammad, From Monolith to Microservice: Measuring Architecture Maintainability, *(IJACSA) International Journal of Advanced Computer Science and Applications*, Vol. 14, No. 5, 2023, <https://doi.org/10.14569/IJACSA.2023.0140591>.
6. G. Blinowski, A. Ojdowska, A. Przybylek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, *IEEE Access*, ISSN 2169-3536, Vol. 10, February 18, 2022, pp. 20357–20374, <https://doi.org/10.1109/ACCESS.2022.3152803>.
7. D. Faustino, N. Gonçalves, M. Portela, A. Rito Silva, Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation, *Performance Evaluation*, Vol. 164, May 2024, <https://doi.org/10.1016/j.peva.2024.102411>
8. J. Lin, M. Sayagh, A. E. Hassan, The Co-evolution of theWordPress Platform and Its Plugins, *ACM Transactions on Software Engineering and Methodology*, Vol. 32, No. 1, Article 19, February 2023, pp. 1–24, <https://doi.org/10.1145/3533700>.
9. I. Lima, J. Candido, M. d'Amorim, Practical detection of CMS plugin conflicts in large plugin sets, *Information and Software Technology*, Vol. 118, Article 106212, pp. 1–13, <https://doi.org/10.1016/j.infsof.2019.106212>.
10. H. Viet Nguyen, C. Kästner, T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications", *ICSE 2014: Proceedings of the 36th International Conference on Software Engineering*, May 31, 2014, pp. 907–918, <https://doi.org/10.1145/2568225.2568300>.
11. N. Wintersgill, T. Stalnaker, L. A. Heymann, O. Chaparro, D. Poshyanyk, “The Law Doesn’tWork Like a Computer”: Exploring Software Licensing Issues Faced by Legal Practitioners, *Proceedings of the ACM on Software Engineering*, Vol. 1, Issue FSE, Article No.: 40, July 12, 2024, pp. 882 - 905, <https://doi.org/10.1145/3643766>.
12. W. Xu, H. He, K. Gao, M. Zhou, Understanding and Remediating Open-Source License Incompatibilities in the PyPI Ecosystem, *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 8, 2023, pp. 178–190, <https://doi.org/10.1109/ASE56229.2023.00175>.
13. F. Neubauer, P. Bredl, M. Xu, K. Patel, MetaConfigurator: A User-Friendly Tool for Editing Structured Data Files. *Datenbank Spektrum* **24**, 161–169 (2024). <https://doi.org/10.1007/s13222-024-00472-7>.
14. F. Neubauer, J. Pleiss, B. Uekermann, *Data Model Creation with MetaConfigurator*, Datenbanksysteme für Business, Technologie und Web (BTW 2025), Gesellschaft für Informatik, Bonn, EISSN: 2944-7682, Student Track. Bamberg. März 2025, pp. 933–944, <https://doi.org/10.18420/BTW2025-60>.
15. F. Neubauer, *Data model creation with MetaConfigurator*, Institute for Visualization and Interactive Systems, University of Stuttgart Universitätsstraße 38 D–70569 Stuttgart, pp. 1–83, <https://doi.org/10.18419/opus-15126>.
16. F. Neubauer, B. Uekermann, J. Pleiss, Data Model Creation with MetaConfigurator, Zenodo (2025), <https://doi.org/10.5281/zenodo.14981537>.
17. P. Ladosz, L. Weng, M. Kim, H. Oh, Exploration in deep reinforcement learning: A survey, *Information Fusion*, Vol. 85, September 2022, pp. 1–22, <https://doi.org/10.1016/j.inffus.2022.03.003>.
18. D. Abel, A. Barreto, B. Van Roy, D. Precup, H. Van Hasselt, A Definition of Continual Reinforcement Learning, 2023, <https://doi.org/10.48550/arXiv.2307.11046>.
19. Y. Song, L. Wei, Q. Yang, J. Wu, L. Xing, Y. Chen, RL-GA: A Reinforcement Learning-based Genetic Algorithm for Electromagnetic Detection Satellite Scheduling Problem, *Swarm and Evolutionary Computation*, Vol. 77, March 2023, Article 101236, pp. 1–29, <https://doi.org/10.1016/j.swevo.2023.101236>.
20. S. Huang, A. Kanervisto, A. Raffin, W. Wang, S. Ontañón, R. F. J. Dossa, A2C is a special case of PPO, May 18, 2022, <https://doi.org/10.48550/arXiv.2205.09123>.