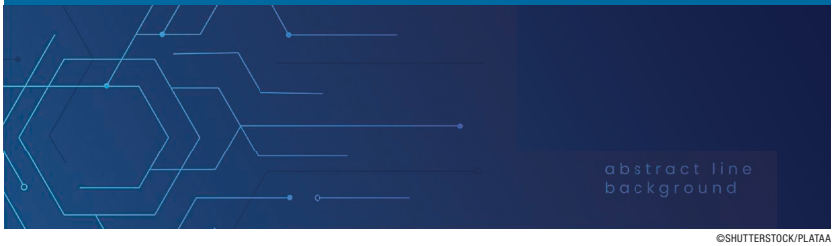
Nautilus: An Interactive Plug-and-Play Search-Based Software Engineering Framework

Thiago Nascimento Ferreira and Silvia Regina Vergilio, Federal University of Paraná

Marouane Kessentini, University of Michigan

// Nautilus Framework allows practitioners to develop and experiment with several multi- and many-objective evolutionary algorithms—guided (or not) by human participation—in a few steps with a minimum required background in coding and search-based algorithms. //



©SHUTTERSTOCK/PLATAA

IT IS A fact that there is a connection between artificial intelligence (AI) and software engineering (SE), which is explored by many works in the literature.[1,2] We can find approaches to solve different SE problems covering the whole software life cycle and derived from all the AI subfields: 1) knowledge representation, reasoning, and decision systems; 2) machine learning; and 3) optimization. The optimization subfield refers to the selection of a best element from some set of available alternatives, which is made based on some performance criteria (objective functions) and a search technique, such as the popular evolutionary algorithms. An example of an SE task to be optimized is to find the minimal set of test cases that satisfies a testing criterion, such as all-branches.

The application of a search technique to solve SE problems is the subject of the search-based SE (SBSE) field.[3] In the last decade, we observed an explosion in the number of SBSE solutions for a great variety of SE tasks. One possible reason for this growth is due to the characteristics of SE problems, which make them more attractive than traditional problems in other engineering disciplines,[4] such as abstraction, directly optimizing the engineering material (e.g., the source code, models, and so on), and the availability of well-defined software metrics that can be optimized.

To ease the creation and implementation of optimization algorithms, reuse techniques, such as application programming interfaces, libraries, design patterns, and frameworks,[5] can be employed, increasing software productivity, decreasing software development and maintenance costs, and improving the software quality by reducing the number of bugs, as the reused part has already been tested and evaluated. For instance, we note some famous frameworks, largely used to implement solutions for problems from different areas: PISA,[6] jMetal,[7] MOEA,[8] ECJ,[9] PlatEMO,[10] DESDEO,[11] and DEAP.[12]

These frameworks contribute to the success and popularity of SBSE solutions for many SE tasks.[4] However, there are some challenges that

Digital Object Identifier 10.1109/MS.2020.3039694
Date of current version: 20 August 2021

need to be addressed to make these solutions more useful for software engineers in real-world settings. One of these main challenges is the lack of user-friendly frameworks that can provide step-by-step support for software engineers during the adoption of existing search algorithms. In fact, it is required that software engineers should have a significant background on optimization to adopt these algorithms for their SE problems, including refactoring, testing, and so on. Furthermore, they may get lost with a lot of details about the parameters' tuning, type of change operators, and solutions representation.

Even worse, the application contexts of SBSE currently encompass a great number of objectives, constraints, and complex inputs as well as outputs. Most SBSE problems are multi- and many-objective, which are not straightforward enough to adopt or navigate through their results. Another practical issue is the usefulness of the solutions generated. Many times, users do not recognize the solutions as good because these ones were not generated considering their needs, preferences, and contexts.

The use of many-objective evolutionary algorithms and the participation of the user (developers, testers, managers, practitioners, and decision makers) in the creation of the SBSE solutions can help solve these challenges. To this end, SBSE approaches should provide different levels of automation, making small decisions and invoking human participation with more fundamental ones.

Most of the existing frameworks do not even have an official user interface with which the user can interact. Although the frameworks are platform independent, not one is integrated with cloud computing, which could allow scalability and its use for large-problem instances. They are not available online as web applications, supporting reports, or user customizations of some interface aspects.

To overcome these limitations, we introduce Nautilus Framework: a free, plug-and-play extendable and open source Java web platform framework that allows user feedback, capturing, developing, and experimenting with several multi- and many-objective evolutionary algorithms. In Nautilus Framework, the users can just "plug" their optimization problems and "play" with the available optimization algorithms. The purpose of Nautilus Framework is to allow SE and AI practitioners to develop their own optimization algorithms to solve their problems—guided (or not) by human participation—by requiring a minimum background in coding and search-based algorithms. Table 1 lists a comparison between Nautilus and other existing frameworks found in the literature with regard to their existing features.

Nautilus Framework Principles

The following principles have guided the development of Nautilus Framework:

- *Simplicity and ease of use*: Nautilus works with jMetal, and as a result, some optimization algorithms provided by jMetal are already available, which can be easily executed and configured via a user-friendly interface. To this end, the user need only select an instance of a configured problem in which he or she is interested. After the execution, the user can visualize and easily choose or evaluate a solution.
- *Portability*: Nautilus is developed in Java, which allows for its execution in machines with different architectures and/or

Table 1. A comparison of existing frameworks.

Feature	Nautilus	PISA	jMetal	MOEA	ECJ	PlatEMO	DESDEO	DEAP
Multiobjective optimization	✓	✓	✓	✓	✓	✓	✓	✓
Cloud support	✓	—	—	—	—	—	—	—
User interface	✓	—	—	✓	—	✓	—	—
Preference support	✓	—	✓	✓	—	✓	✓	—
Web application	✓	—	—		—	—	—	—
User customization	✓	—	—	✓	—	✓	—	—
Pareto-front visualization	✓	—	—	✓	—	✓	—	—

its running in distinct operating systems.

- *Extensibility*: New optimization algorithms, search operators, and optimization problems should be easily added. To reach this principle, Nautilus supports plug-ins in which the users can adapt their needs or context to the tool.
- *Performance and scalability*: Nautilus is a web platform application that executes in cloud computing. This last characteristic allows for automatic software updates, mobility, performance, and scalability. For instance, it is possible to read large-problem instances by splitting them in multiple small subroutines and calculate objective functions or run multiple algorithms in parallel.
- *Customizability*: Nautilus provides a multiuser system in which each user can customize some information and upload to the tool his or her own problem instances to be optimized. Also, the users are able to change some information they visualize about the found solutions and customize some interface features.

Many-Objective Algorithms

Diverse SE problems are many-objective, that is, they are impacted by more than three objectives. The prioritization of test cases is one example, as it is impacted by different factors such as cost, size of the test set to be used, and the ability to reveal faults; code coverage; and so on. To deal with such problems, different many-objective evolutionary algorithms exist. They can be classified into distinct categories[13] according to the strategy implemented to deal with the large/exponential number of nondominated solutions, which are possible for multiobjective problems. For instance, the following are categories of algorithms supported by Nautilus Framework: 1) NSGA-III, an algorithm that uses the concept of reference sets; 2) R-NSGA-II, a preference-based algorithm that reduces the number of solutions by working with a region of interest provided by the user; and 3) PCA-NSGA-II, an algorithm based on dimensionality reduction that reduces the number of solutions by discarding some redundant and nonconflicting objectives.

However, the user is able to extend the framework and implement his or her preference-based, or dimensionality-reduction algorithms. In addition to this, the user can extend Nautilus and implement mechanisms to combine the aforementioned strategies.

Architecture

Nautilus Framework has some nonmodifiable classes that provide a predefined behavior and other ones that can be extended to provide some new functionalities.[14] The first classes belong to the Nautilus Core module, and the last classes belong to Nautilus-Plug-In. Both modules are represented in Figure 1, which contains the Nautilus architecture.

Nautilus uses three main third-party libraries: the jMetal framework, as mentioned previously, for the optimization algorithms; MongoDB, a general-purpose and document-based database; and Spring Boot, a web application framework and inversion of the control container for the Java platform. Besides Nautilus Core, and Nautilus-Plug-In, Nautilus has a third module, called *Nautilus Web*. All of these are briefly described as follows.

Nautilus Core is the most important module because it contains the base classes required by the other modules. For instance, it provides the classes responsible for defining the encoding type of the problems supported, such as binary, integer, and double encoding solutions. The current version of Nautilus Core uses the jMetal implementation for generating solutions; however, in future versions, we plan to release the capability of connecting this module to other optimization frameworks.

Nautilus-Plug-In is responsible for providing extensible classes in which the user can create his or her own plug-ins for Nautilus and adapt his or her needs to the tool. For instance, the user can extend and create new optimization algorithms, optimization problems, mating operators, quality indicators, and preference mechanisms (those provided in the loop).

Nautilus Web is the module responsible for providing a user interface based on a web platform. Through this interface, it is possible to execute the algorithms, visualize the found solutions, interact with the tool, and provide the user preferences and feedback about the solutions. This module uses Nautilus Core and Nautilus-Plug-In and is developed in Spring Boot by utilizing MongoDB to save all of the generated solutions in a database.

Extending Nautilus Framework

As mentioned previously, the classes of Nautilus-Plug-In can be instantiated to add a new problem to be solved as well as new algorithms, including the preference-based ones. In this section, we present an example of extension for the variability testing of software product lines (VTSPLs) problem.[15] This problem refers to the selection of the best set of products to be tested that can be derived from the SPL. The selection can take into account many factors:

the size of the set; and the cost, product similarity, pairwise coverage, and possible faults.

Instantiating a New Problem

We instantiate the VTSPLs problem in Nautilus considering seven objective functions. To implement this problem, it is necessary to extend some classes such as *AbstractProblemExtension*, *AbstractObjective*, and *Instance*. Some of these implementations are described as follows.

Algorithm 1 shows the code of the *AbstractProblemExtension* class. This class is one of the most important classes during the problem instantiation process. In this one, the user can define which encoding type the addressed optimization problem supports, the class responsible for reading an instance file (a file with required information to calculate the objective functions), and the objective functions to be optimized. In this algorithm, SPL testing supports a binary encoding, the instance file is in .txt format, and the objective functions are: Number of Products, Alive Mutants, Uncovered Pairs, Similarity, Cost, Unselected and Unimportant Features.

To calculate each one of the objectives of the VTSPLs problem, the user needs to provide the corresponding implementation and extend the *AbstractObjective* class. Algorithm 2 presents an example of extension to calculate the number of products objective function.

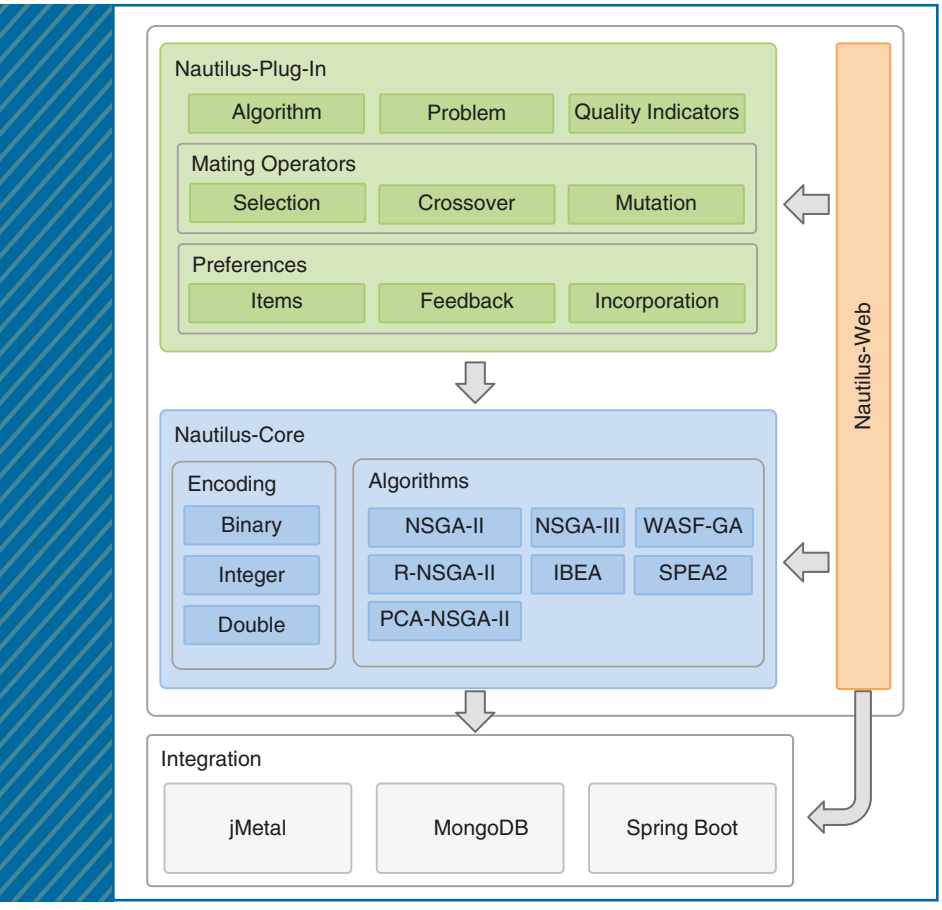


FIGURE 1. The Nautilus Framework architecture.

```
1  @Extension
2  public class SPLProblemExtension
3      extends AbstractProblemExtension {
4
5      @Override
6      public Problem<?> getProblem
           (Instance in,
7        List<AbstractObjective> obj) {
8        return new VTSPLProblem(in, obj);
9      }
10
11     @Override
12     public String getName() {
13       return "VTSPL Problem";
14     }
15
16     @Override
17     public Class<? Extends Solution<?>>
18       supports() {
19       return BinarySolution.class;
20     }
21
22     @Override
23     public List<AbstractObjective>
                 getObjectives() {
24       return Arrays.asList(
25           new NumberOfProducts(),
26           new AliveMutants(),
27           new UncoveredPairs(),
28           new NewSimilarity(),
29           new Cost(),
30           new UnselectedFeatures(),
31           new UnimportantFeatures()
32       );
33     }
34
35     @Override
36     public Instance getInstance(Path path){
37       return new TXTInstanceData(path);
38     }
39 }
```

Algorithm 1. Instantiating the VTSPLs problem.

The user must define a name for the desired objective function and the corresponding implementation. As a default behavior, Nautilus considers that an objective function must be minimized. However, this default behavior can be changed.

Regarding the *Instance* class, this one is responsible for saving information read from the input file (used as a problem instance). This information is used for evaluating the solutions generated.

Instantiating a New Algorithm

If the addressed problem requires an optimization algorithm different from those already implemented in Nautilus, the user must extend the *AbstractAlgorithmExtension* class. To illustrate this, Algorithm 3 shows an extension in which the SPEA2 algorithm, available in jMetal, is added.

In this way, we can instantiate Nautilus by extending the classes with implementation of different algorithms. But a preference-based algorithm requires a different mechanism to provide or incorporate user preferences. To this end, the user can extend the *AbstractPreferenceExtension* class, as described in Algorithm 4.

In the example, the user is required to provide feedback for some solutions by using an ordinal scale composed of items *Not preferred*, *No Opinion*, and *Preferred*. The feedback is provided interactively and incorporated into the objective functions by weighting them to the next execution.

Once the required basic classes are extended, the user can generate a final plug-in file and upload it to Nautilus by using the GUI provided for this purpose.

Using the VTSPLs Extension

In this section, we present some Nautilus screenshots that illustrate the use of the VTSPLs extension. First, the user should sign up and log in to the system. Then, the user can see information about all the executions already performed and those that are in execution. In Nautilus, an execution is associated with an algorithm and its corresponding parameters and with the Pareto front, which is composed of the obtained nondominated solutions.

```
1  public class NumberOfProductsObjective
2      extends AbstractObjective {
3
4      protected int selectedProducts;
5
6      @Override
7      public void beforeProcess(Instance i,
               Solution<?> s) {
8          this.selectedProducts = 0;
9      }
10
11     @Override
12     public void process(Instance i,
               Solution<?> s, int id) {
13         selectedProducts++;
14     }
15
16     @Override
17     public double calculate(Instance
               i, Solution<?> sol) {
18         return selectedProducts/
               i.NumberOfProducts();
19     }
20
21     @Override
22     public String getName() {
23         return "Number Of Product";
24     }
25 }
```

Algorithm 2. Instantiating an objective function.

To start a new execution, the user needs to choose the problem instance to be optimized and set the algorithm parameters, such as mating operators, number of evaluations, and population size as well as to specify the number of runs for this setting. Once the optimization

```
1  @Extension
2  public class SPEA2AlgorithmExtension
3      extends AbstractAlgorithmExtension {
4
5      @Override
6      public Algorithm<? extends
               Solution<?>>
7          getAlgorithm(Builder builder) {
8          return new SPEA2(builder);
9      }
10
11     @Override
12     public String getName() {
13         return "SPEA2";
14     }
15 }
```

Algorithm 3. Instantiating the SPEA algorithm.

```
1  @Extension
2  public class ConfidencePreferenceExtension
3  extends AbstractPreferenceExtension {
4
5      @Override
6      public AbstractFeedback get
               Feedback() {
7          return new OrdinalScale();
8      }
9
10     @Override
11     public AbstractIncorporation
12         getIncorporation() {
13         return new WeightedGuidance();
14     }
15 }
```

Algorithm 4. Instantiating user preferences.

is done, the user can observe the solutions (Figure 2), either by using a chart or a table, both of which contain the objective values.

Another important feature of this page is customization. It is possible to change some of the displayed information (such as the chart color), remove duplicated solutions from the Pareto front, and normalize objective values. So, to open and visualize a solution, simply click on the circle in the chart. In this example, solution number 60 was selected and, as a result, Nautilus presents information about the selected solution, as illustrated in Figure 3.

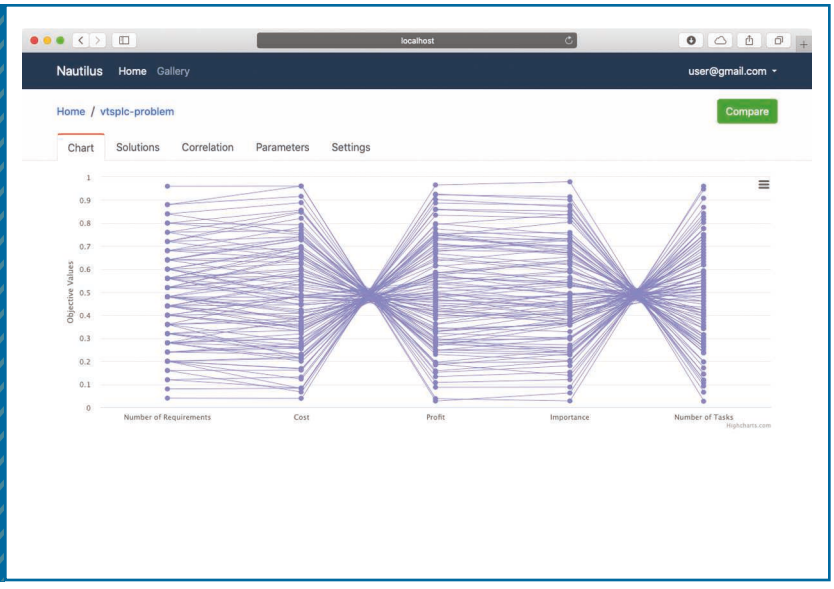


FIGURE 2. The Nautilus execution page.

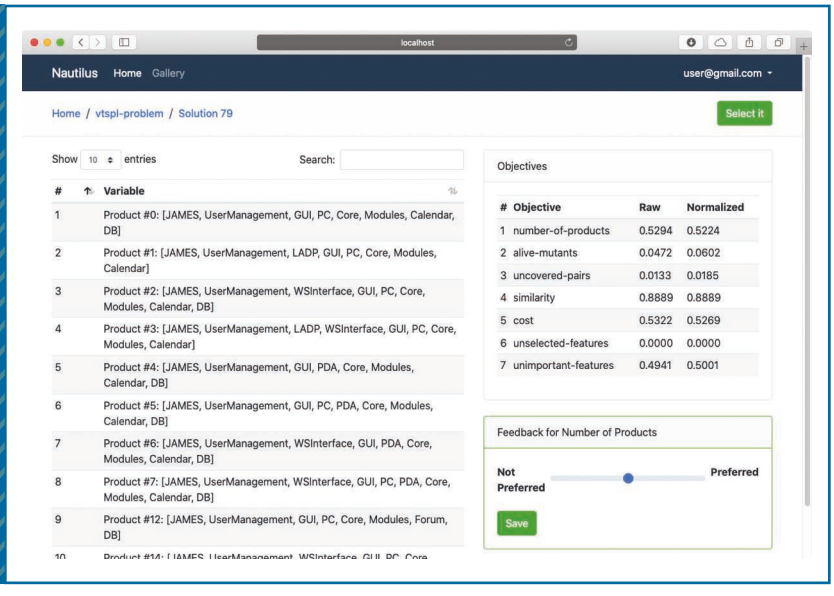


FIGURE 3. The Nautilus solution page.

It is possible to see the variables from the selected solution and the corresponding raw and normalized objective values. Also, users can provide their preferences about the solutions by just sliding left or right the component below the objective values. In this example, the user provides a feedback *Not Preferred*, *No Opinion*, and *Preferred* to the visualized solution. Again, this component can be changed by extending specific classes from Nautilus-Plug-In as well as the kind of information provided by the user. If the user considers the solution good, her or she can click on the "Selected" button to end the search.

Evaluating Nautilus Framework

Using the VTSPLs problem, we conducted an evaluation with a group of 12 potential users of Nautilus. This group was composed of practitioners with different skills and experiences with SPL, software testing, and optimization algorithms, of which 12 are currently Ph.D. students and five have experience with software development in companies. The participants' years of experience in programming ranged, in general, from two to 10.

Each participant executed a set of different optimization algorithms, including algorithms based on preferences provided interactively. In the end, they were asked to select a solution they considered good. After the experiment, each participant completed a questionnaire that evaluated their experience with using Nautilus. The results are described as follows.

Regarding the time spent to get familiar with Nautilus eight participants

(66.7%) took fewer than 10 min, during which five of them spent fewer than 5 min. In addition, all users claimed to have spent fewer than 10 min to explore the Pareto front by using visualization support.

Figure 4 shows that seven participants (58.3%) said it was easy to learn how to operate the tool and just one claimed difficulty. Moreover, nine participants (75%) stated it was easy to understand the task they were asked to do. Still in this context, 10 participants (83.3%) asserted it was easy to locate and identify relevant solutions. In addition, 50% of the users stated it was easy to use visualization support for the Pareto front, while the other 50% claimed it was neither easy nor difficult. A total of eight participants asserted that Nautilus had a user-friendly interface, that the navigation was very easy, and that the error messages were helpful. We also asked the users their opinions about the organization of the information in the screen. In this case, five participants (41.7%) stated the information on the screen was clear, four (33.3%) stated the information was very clear, and just three participants (25%) chose the neutral option.

The users also opined about the best features provided by Nautilus. Figure 5 presents the results. For most users, the best feature Nautilus provided was its Pareto-front visualization, followed by its interface and cloud-computing support.

To better evaluate the users' opinions about the features provided by Nautilus in comparison with existing frameworks, we asked users with previous experience with other frameworks to provide agreement rates about each feature listed in Table 1. The results are shown in Figure 6. For most of the features, Nautilus provides better support in comparison

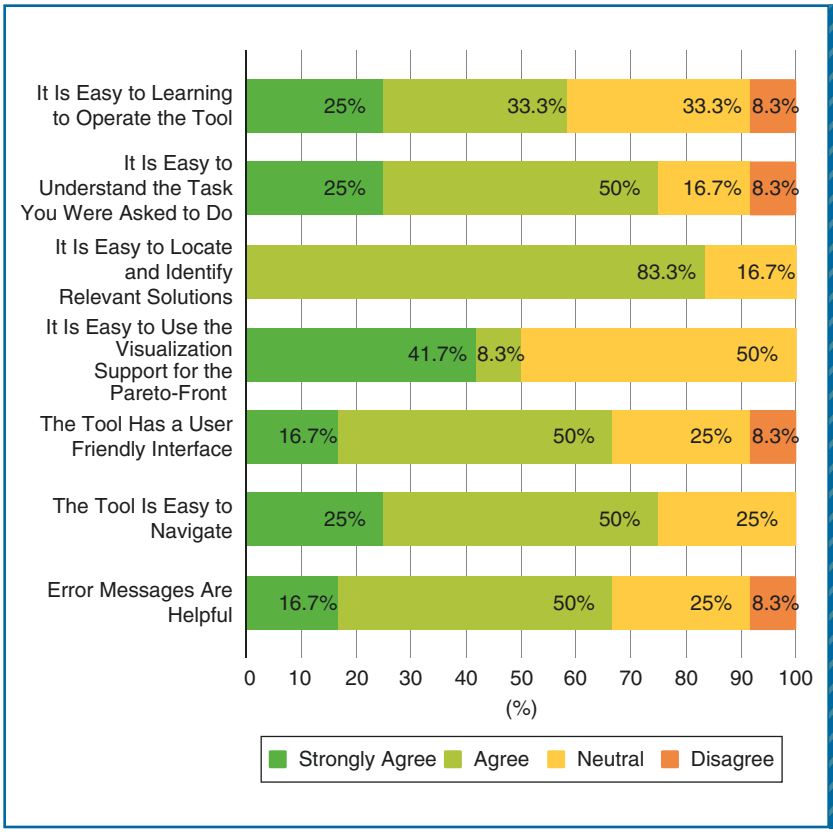


FIGURE 4. Users' feedback.

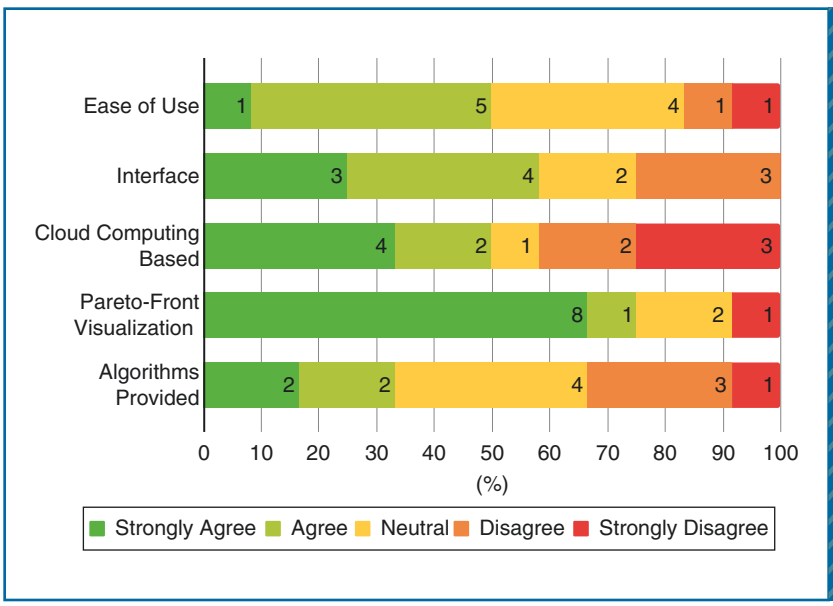


FIGURE 5. Nautilus' best features.

with existing frameworks. For the latter, the users pointed out the lack of support for user customization and web applications and the existence of a user interface.

We also provided open questions, allowing users to write about Nautilus advantages and disadvantages in comparison to other frameworks they had previously used. Most of them pointed out as an advantage the support to Pareto-front visualization and user interaction; for instance, that the user can set some parameters aiming to improve the solutions generated. As a disadvantage, the users mentioned the lack of more optimization algorithms and a history of user actions. We intend to address these limitations in a future version of Nautilus.

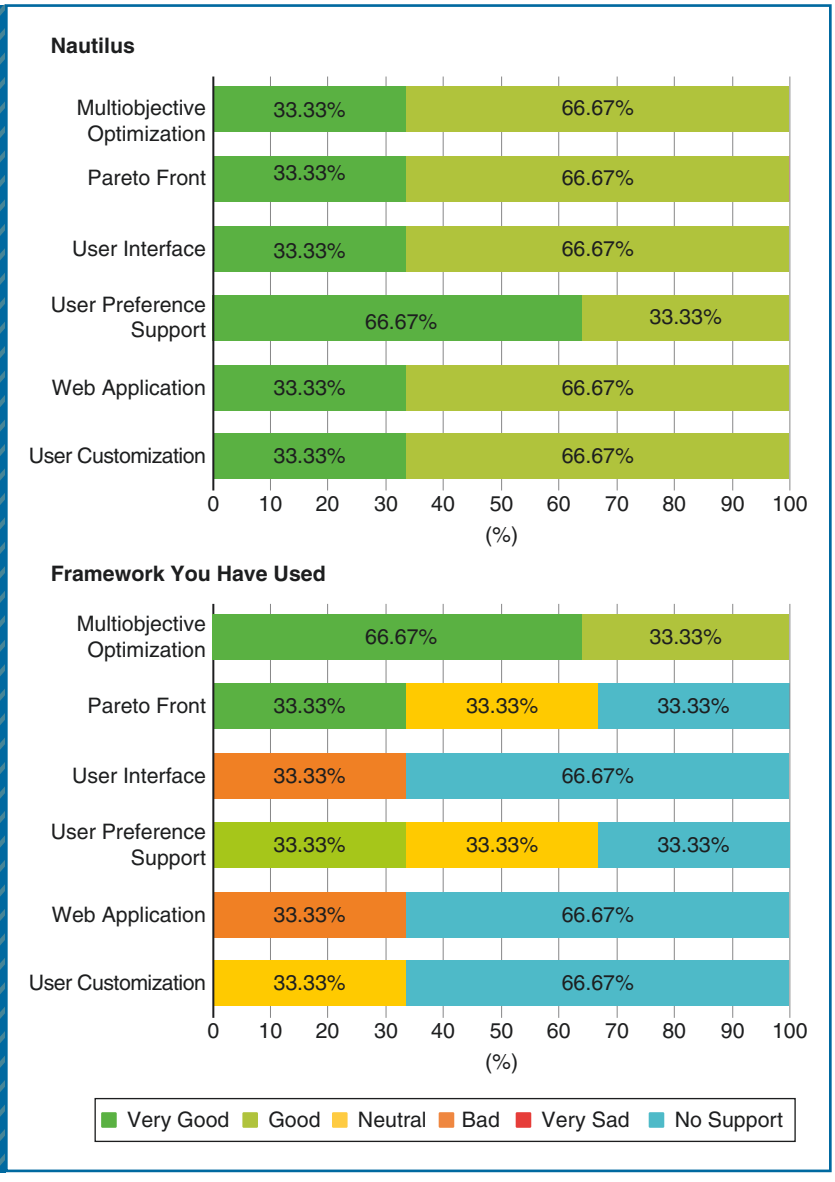


FIGURE 6. The agreement rates regarding the features provided by Nautilus and existing frameworks.

This article introduced Nautilus Framework, a plug-and-play extendable and Java web-based framework for many-objective optimization with human participation. Nautilus has the following main features:

- plug-ins to allow extensibility
- the instantiation of different problems to be optimized and extension for implementing search operators and many-objective functions
- the use of different optimization algorithms, with an emphasis on many-objective ones from the categories based on user preferences, Pareto dominance, reference set, and dimensionality reduction; some of these algorithms and mating operators are available in the framework, and new ones can also be extended
- a user-friendly interface that allows for visualizing solutions and their objective values, capturing user feedback, and customization (that is, color, language, and decimal separators and places)
- the calculation of some quality indicators widely used in the literature, such as hypervolume and inverted generational distance, and other ones for preference-based algorithms, such as hypervolume based on R-metric and inverted generational distance based on R-metric
- A web-based platform that allows for scalability. The framework can run in cloud computing, supporting

ABOUT THE AUTHORS

THIAGO NASCIMENTO FERREIRA is with the Federal University of Paraná (UFPR), Curitiba, 81.531-980, Brazil. His main research interests are bioinspired computation, multiobjective optimization, and preference-based optimization algorithms focused on search-based software engineering. Ferreira received his Ph.D. in computer science from UFPR in 2019. Contact him at tnferreira@inf.ufpr.br.

SILVIA REGINA VERGILIO is a professor of software engineering (SE) in the computer science department of the Federal University of Paraná, Curitiba, 81.531-980, Brazil, where she leads a research group on SE. Her research interests include software testing, software reliability, software product lines, and search-based SE (SBSE). She serves as the assistant editor of *Journal of Software Engineering: Research and Development* and acts as a peer reviewer for diverse international journals. She serves on the program committee of numerous conferences related to SBSE and software testing. Contact her at silvia@inf.ufpr.br.

MAROUANE KESSENTINI is a tenured associate professor and leads a research group on software engineering (SE) intelligence. Kessentini received his Ph.D. from the University of Montréal, Québec, Canada, in 2012. He is a recipient of the prestigious 2018 Distinguished Research Award from the President of Tunisia, the University Distinguished Teaching (2017) Award, the University Distinguished Digital Education (2018) Award, the College of Engineering and Computer Science Distinguished Research (2018) Award, and four Best Paper Awards. He has received several grants from both industry and federal agencies and has published more than 110 papers in top journals and conferences. Contact him at marouane@umich.edu.

optimization problems with large instances and many numbers of objectives.

An open source implementation of Nautilus Framework is available at https://github.com/nautilus-framework. The application of optimization algorithms generate solutions that have been proved to increase the effectiveness and efficiency of many SE tasks. Nautilus Framework contributes to fulfilling new demands required by today's software applications, allowing the implementation of adaptive solutions, considering real and many-objective scenarios, and including user participation in an interactive way. The main Nautilus features allow support to the construction of AI solutions guided by human decisions. As a future work, we intend to extend Nautilus to work with other optimization frameworks available in the literature.

Acknowledgments

This work is supported by CAPES and CNPq grants 307762/2015-7 and 473899/2013-2. The corresponding author is Thiago Nascimento Ferreira.

References

1. M. Harman, "The role of artificial intelligence in software engineering," in *Proc. 1st Int. Workshop Realizing AI Synergies Softw. Eng. (RAISE)*, June 2012, pp. 1–6.
2. L. Ford, "Artificial intelligence and software engineering: A tutorial introduction to their relationship," *Artif. Intell. Rev.*, vol. 1, no. 4, pp. 255–273, Dec. 1987. doi: 10.1007/BF00142926.
3. M. Harman and B. F. Jones, "Search-based software engineering," *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, Dec. 2001. doi: 10.1016/S0950-5849(01)00189-6.
4. M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–61, 2012. doi: 10.1145/2379776.2379787.
5. A. V. Tsyganov and O. I. Bulychov, "Implementing parallel metaheuristic optimization framework using metaprogramming and design patterns," in *Information Technology Applications in Industry* (Applied Mechanics and Materials), vol. 263, J. Zhang, Z. Wang, S. Zhu, and X. Meng, Eds. Switzerland: Trans Tech Publications Ltd., Feb. 2013, pp. 1864–1873.
6. S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA – A platform and programming language independent interface for search algorithms,"

in *Proc. 2nd Int. Conf. Evolutionary Multi-Criterion Optimiz. (EMO '03)*. Faro, Portugal: Springer-Verlag, 2003, pp. 494–508.

7. J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Adv. Eng. Softw.*, vol. 42, no. 10, pp. 760–771, Oct. 2011. doi: 10.1016/j.advengsoft.2011.05.014.
8. D. Hadka, "MOEA Framework: A free and open source Java framework for multiobjective optimization. User manual," 2016. Accessed: Aug. 20, 2019. [Online]. Available: http://www.moeaframework.org/
9. E. O. Scott and S. Luke, "ECJ at 20: Toward a general metaheuristics toolkit," in *Proc. Genetic Evolutionary Comput. Conf. Companion (GECCO '19)*, 2019, pp. 1391–1398.
10. Y. Tian, R. Cheng, X. Zhang, and Y. Jin, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization," *IEEE Comput. Intell. Mag.*, vol. 12, no. 4, pp. 73–87, 2017. doi: 10.1109/MCI.2017.2742868.
11. V. Ojalehto and K. Miettinen, "DESDEO: An open framework for interactive multiobjective optimization," in *Multiple Criteria Decision Making and Aiding*. H. Sandra, M. J. Geiger, and A. T. de Almeida, Eds. New York: Springer-Verlag, 2019, pp. 67–94.
12. F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Learn. Res.*, vol. 13, pp. 2171–2175, July 2012.
13. B. Li, J. Li, K. Tang, and X. Yao, "Many-objective evolutionary algorithms: A survey," *ACM Comput. Surv.*, vol. 48, no. 1, p. 13, Sept. 2015. doi: 10.1145/2792984.
14. T. N. Ferreira, S. R. Vergilio, and M. Kessentini, "Many-objective search-based selection of software product line test products with Nautilus," in *Proc. 24th Int. Syst. Softw. Prod. Line Conf. (SPLC '20) –Demo Track*, 2020, pp. 1–4. doi: 10.1145/3382026.3431248.
15. T. N. Ferreira, J. A. P. Lima, A. Strickler, J. N. Kuk, S. R. Vergilio, and A. Pozo, "Hyper-heuristic based product selection for software product line testing," *IEEE Comput. Intell. Mag.*, vol. 12, no. 2, pp. 34–45, May. 2017. doi: 10.1109/MCI.2017.2670461.

Digital Object Identifier 10.1109/MS.2021.3099679