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

Thiago do Nascimento Ferreira[1] **and Silvia Regina Vergilio**
Federal University of Paraná

Marouane Kessentini
University of Michigan

***Abstract*—Several Software Engineering problems are complex and encompass a great number of objectives to be handled. However, practitioners may face several challenges to adopt existing metaheuristic search for their problems due to the lack of background, or some difficult choices such as the change operators, and parameters tuning. *Nautilus Framework* allows practitioners developing and experimenting several multi- and many-objectives evolutionary algorithms guided (or not) by human participation in few steps with a minimum required background in coding and search-based algorithms. A case study illustrates its benefits, which can also be used to support the construction of AI solutions guided by human decisions.**
Keywords: preference and search based software engineering, many-objective optimization, plugin and play framework.

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], [3], [4], [5]. We can find approaches to solve different SE problems covering the whole software life cycle, derived from all the AI sub-fields: (a) Knowledge Representation, reasoning and Decision systems (KR&D); (b) Machine Learning (ML); and (c) Optimization. The Optimization sub-field 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 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 Software Engineering (SBSE) field [6], [7], [8]. We observed in the last decade an explosion in the number of SBSE solutions for a great variety of SE tasks. One possible reason of this growth is due to the characteristics of the SE problems that make them more attractive than the traditional problems in other engineering disciplines [9] such as the abstraction, optimizing directly the engineering material (e.g. source code, models, etc.) and the availability of well-defined software metrics that can be optimized.

To ease the creation, and implementation of

[1]Corresponding Author. E-mail: tnferreira@inf.ufpr.br.

optimization algorithms, reuse techniques, such as APIs, libraries, design patterns and frameworks [10], can be employed, increasing software productivity, decreasing software development and maintenance costs, and improving the software quality by reducing the number of bugs, since the reused part has already been tested and evaluated. For instance, we can mention some famous frameworks, largely used to implement solutions for problems from different areas: PISA framework [11], jMetal [12], MOEA Framework [13], ECJ [14], PlatEMO [15], DESDEO [16], and DEAP [17].

These frameworks contribute to the success and popularity of SBSE solutions for many SE tasks [9]. However, we can identify some challenges that need to be addressed to make these solutions more useful for software engineers in a real-world setting. 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 significant background on optimization to adopt these algorithms for their SE problems including refactoring, testing, etc. 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-objectives which are not straightforward to adopt or navigate through their results. Another practical issue is the usefulness of the solutions generated. Many times the 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 to more fundamental ones.

Most of the existing frameworks do not have even an official user interface in which the user can interact. Although the frameworks are platform-independent, no one is integrated with cloud computing, which could allow scalability and its use for large problem instances. They are not available online as a web application, supporting reports, user customization 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 shows a comparison between *Nautilus* and other existing frameworks found in the literature regarding the features that they have.

Table 1. Comparison among existing frameworks.

Feature	*Nautilus*	PISA	jMetal	MOEA	ECJ	PlatEMO	DESDEO	DEAP
Multi-objective optimization	✓	✓	✓	✓	✓	✓	✓	✓
Cloud Support	✓							
User Interface	✓			✓		✓		
Preference Support	✓		✓	✓		✓	✓	
Web Application	✓							
User Customization	✓			✓		✓		
Pareto-front visualization	✓			✓		✓		

Nautilus Framework Principles

The following principles guided the development of *Nautilus Framework*:

- **Simplicity and easy-to-use:** *Nautilus* works with jMetal. Then some optimization algo-

rithms provided by jMetal are already available, which can be easily executed and configured via a user-friendly interface. To this end, the user needs only to select an instance of a configured problem s(he) is interested. After the execution, the user can visualize and easily choose or evaluate a solution;

- **Portability:** *Nautilus* is developed in Java, which allows its execution in machines with different architectures and/or running in distinct operating systems;
- **Extensibility:** New optimization algorithms, search operators, and optimization problems should be easily added. To reach this principle, *Nautilus* supports plugins 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 automatic software updates, mobility, performance, and scalability. For instance, it is possible to read large problem instances by splitting them in multiple small sub-routines, and calculate the objective functions or run multiple algorithms in parallel;
- **Customizability:** *Nautilus* provides a multi-user system in which each user can customize some information and upload to the tool his/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, impacted by more than three objectives. The prioritization of test cases is an example, impacted by different factors such as cost, size of the test set to be used, 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 in distinct categories [18] according to the strategy implemented to deal with the large/exponential number of non-dominated solutions, which are possible for multi-objective problems. For instance, we can mention the following categories of algorithms supported by *Nautilus Framework*: i) NSGA-III: is an algorithm that uses the concept of Reference Set; ii) R-NSGA-II and WASF-GA are preference-based algorithms that reduces the number of solutions by working with a region of interest provided by the user; iii) PCA-NSGA-II is an algorithm based on dimensionality reduction that reduces the number of solutions by discarding some redundant and non-conflicting objectives; iv) IBEA, an algorithm based on indicators; and v) SPEA2 is based on the concept of Pareto domination and external archiving.

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

Architecture

Nautilus Framework has some non-modifiable classes that provide a pre-defined behavior, and other ones that can be extended to provide some new functionalities [19]. The first classes belong to the module *Nautilus-Core* and the last to *Nautilus-Plugin*. Both modules are represented in Figure 1 that contains the *Nautilus* architecture.

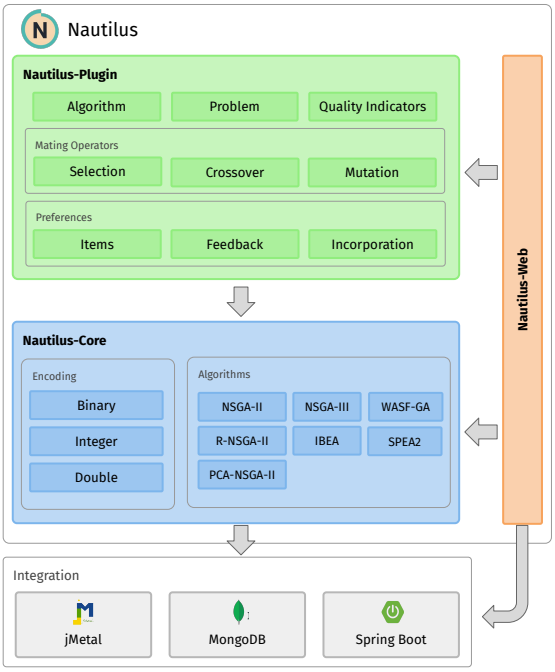

Figure 1. *Nautilus Framework Architecture*

Nautilus uses three main third-party libraries: the jMetal framework, as mentioned before, for the optimization algorithms; MongoDB, a

general purpose and document-based database; and Spring Boot, a web application framework and inversion of control container for the Java platform. Besides *Nautilus-Core*, and *Nautilus-Plugin*, *Nautilus* has a third module, called *Nautilus-Web*. All of them 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 jMetal implementation for generating solutions. It is responsible for i) providing the set of optimization algorithms; ii) the basic solution representations; and iii) some quality attributes, while Nautilus. However, we plan to release in future versions the capability of connecting this module to other optimization frameworks. Besides, this module provides a mechanism to generate an approximation to the True Pareto-front putting together all solutions generated in multiple runs (for a given problem and instance, considering or not different algorithms) and removing repeated and non-dominated ones.

Nautilus-Plugin is responsible for providing extensible classes in which the user can create his/her own plugins for *Nautilus* and adapt his/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 ones in-the-loop provided).

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, provide the user preferences and feedback about the solutions. This module uses *Nautilus-Core* and *Nautilus-Plugin* and is developed in Spring Boot by using MongoDB for saving in a database all generated solutions.

Extending *Nautilus Framework*

As mentioned before, the classes of *Nautilus-Plugin* 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 show an example of extension for the Variability Testing of Software Product Lines (VTSPL) problem [20]. 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: size of the set, cost, product similarity, pairwise coverage, and possible faults.

Instantiating a New Problem

We instantiate the VTSPL 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 class 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 figure, the 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.

```
@Extension
public class SPLProblemExtension
    extends AbstractProblemExtension {

    @Override
    public Problem<?> getProblem(Instance
    in,
      List<AbstractObjective> obj) {
      return new VTSPLProblem(in, obj);
    }

    @Override
    public String getName() {
      return "VTSPL Problem";
    }

    @Override
    public Class<? extends Solution<?>>
      supports() {
      return BinarySolution.class;
    }

    @Override
    public List<AbstractObjective>
    getObjectives() {
        return Arrays.asList(
            new NumberOfProducts(),
            new AliveMutants(),
```

```
            new UncoveredPairs(),
            new NewSimilarity(),
            new Cost(),
            new UnselectedFeatures(),
            new UnimportantFeatures()
        );
    }

    @Override
    public Instance getInstance(Path path){
      return new TXTInstanceData(path);
    }
}
```

Algorithm 1. Instantiating VTSPL problem

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

```
public class NumberOfProductsObjective
    extends AbstractObjective {

    protected int selectedProducts;

    @Override
    public void beforeProcess(Instance i,
    Solution<?> s) {
        this.selectedProducts = 0;
    }

    @Override
    public void process(Instance i,
    Solution<?> s, int id) {
        selectedProducts++;
    }

    @Override
    public double calculate(Instance i,
    Solution<?> sol) {
        return selectedProducts / i.
    NumberOfProducts();
    }

    @Override
    public String getName() {
        return "Number Of Products";
    }
}
```

Algorithm 2. Instantiating an objective function

Regarding the *Instance* class, this one is responsible for saving information read from the input file (used as 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 ones 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.

```
@Extension
public class SPEA2AlgorithmExtension
    extends AbstractAlgorithmExtension {

    @Override
    public Algorithm<? extends Solution<?>>
        getAlgorithm(Builder builder) {
        return new SPEA2(builder);
    }

    @Override
    public String getName() {
        return "SPEA2";
    }
}
```

Algorithm 3. Instantiating the SPEA algorithm

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.

```
@Extension
public class ConfidencePreferenceExtension
extends AbstractPreferenceExtension {

    @Override
    public AbstractFeedback getFeedback() {
        return new OrdinalScale();
    }

    @Override
    public AbstractIncorporation
        getIncorporation() {
        return new WeightedGuidance();
    }
}
```

Algorithm 4. Instantiating user preferences

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 in the objective functions by weighting them to the next execution.

Once the required basic classes are extended, the user can generate a final plugin file and upload

it to *Nautilus* by using the graphical interface provided to this purpose.

Using the VTSPL Extension

In the next paragraphs, we present some *Nautilus*'s screenshots that illustrate the use of the VTSPL extension.

First, the user should sign up and log in the system. Then, the user can visualize information about all executions already performed and those ones in execution. In *Nautilus*, an execution is associated with an algorithm and corresponding parameters, and with the Pareto-front composed by the obtained non-dominated solutions.

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 is done, the user can visualize the solutions (Figure 2), either by using a chart or a table, which contains the objective values.

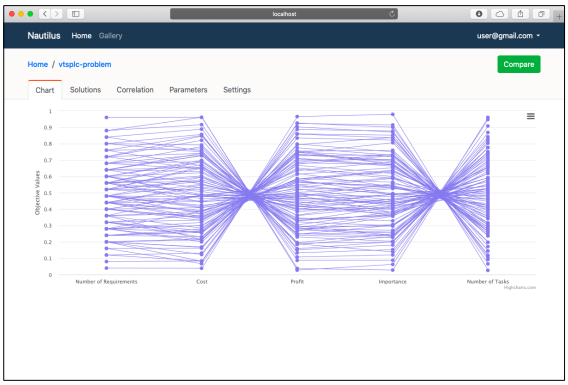

Figure 2. Execution page

Another important feature in this page is customization. It is possible to change some displayed information such as the chart color, remove duplicated solutions from Pareto-front, and normalize objective values. So, to open and visualize a solution, it is necessary just to click in the circle on the chart.

In this example, Solution #79 was selected and, as a result, *Nautilus* presents information about the selected solution as illustrated in Figure 3.

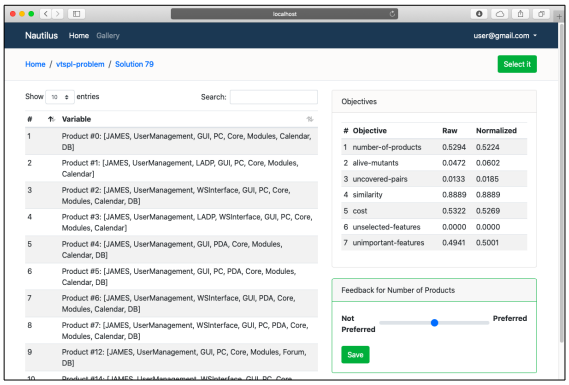

Figure 3. Solution page

It is possible to see the variables from the selected solution and 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-Plugin*, as well as the kind of information provided by the user. If the user considers the solution good, s(he) can click on the button "Selected" to end the search.

Evaluating *Nautilus Framework*

Using the VTSPL problem, we conducted an evaluation with a group of 12 potential users of *Nautilus*. This group is composed by practitioners with different skills and experiences on SPL, software testing, and optimization algorithms, in which 7 are currently Ph.D. students, and 5 have experience with software development in companies. The experience in years of these participants on programming ranged, in general, from 2 to 10 years.

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 answered a questionnaire aiming to evaluate their experience using *Nautilus*. The results are described as follows.

Regarding the time spent to get familiar with *Nautilus*, 8 participants (66.7%) took less than 10 minutes in which 5 of them spent less than 5

minutes. Besides, all users claimed to spend less than 10 minutes to explore the Pareto-front by using the visualization support.

Figure 4 presents that 7 participants (58.3%) said it was easy to learn how to operate the tool and just one claimed difficulty. Besides, 9 participants (75%) stated that it was easy to understand the task they were asked to do. Still in this context, 10 participants (83.3%) asserted that it was easy to locate and identify relevant solutions. Besides, 50% of the users stated it is easy to use the visualization support for the Pareto-front. The other 50% claimed it was neutral. A total of 8 participants asserted that *Nautilus* has a user-friendly interface, that the navigation is very easy and the error messages are helpful. We also asked to the users their opinions about the organization of the information in the screen. In this case, 5 participants (41.7%) stated that the information in the screen is clear, 4 participants (33.3%) stated that the information is very clear, and just 3 participants (25%) chose the neutral option.

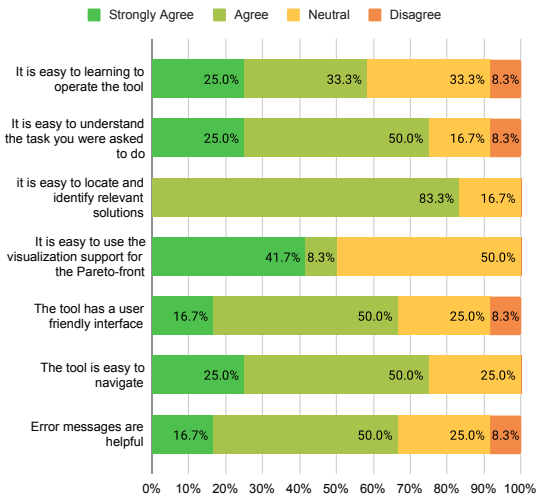

Figure 4. Users' feedback

The users also opined about the best features provided by *Nautilus*. Figure 5 shows the results. For most users, the best feature *Nautilus* provides is the Pareto-front visualization followed by the interface.

To better evaluate the users' opinion about the features provided by *Nautilus* in comparison with existing frameworks, we asked users, with previous experience with other frameworks, to

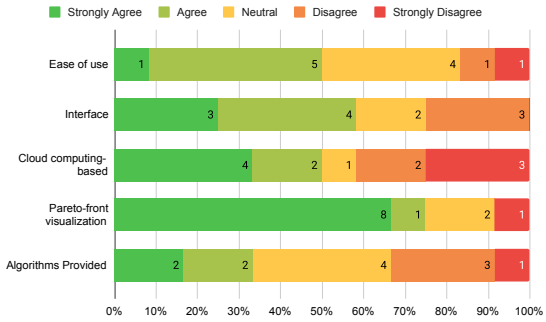

Figure 5. Best features.

provide agreement rates about each feature of Table 1. The results are shown in Figure 6. For most features, *Nautilus* provides better support in comparison with existing frameworks. For the latter, the users pointed out the lack of support for user customization, web application and existence of a user interface.

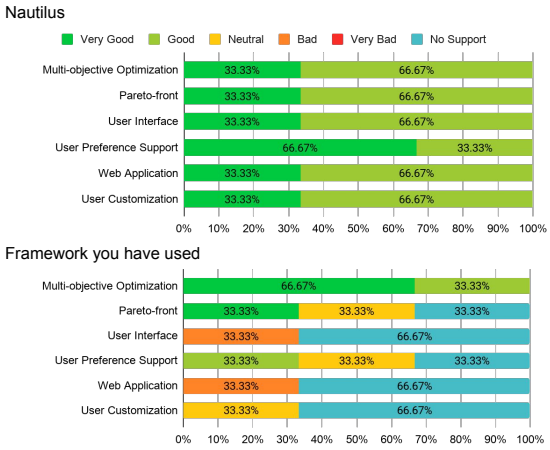

Figure 6. Agreement rates regarding features provided by *Nautilus* and existing frameworks.

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

Conclusion

This work 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:

- Plugins to allow extensibility;
- Instantiation of different problems to be optimized, and extension for implementing search operators and many-objective functions;
- Use of different optimization algorithms, with an emphasis in 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 visualizing the solutions and their objective values, capturing user feedback, and can be customized (that is, color, language, decimal separator and places);
- Calculation of some quality indicators widely used in the literature, such as hypervolume and IGD, and other ones for preference-based algorithms such as R-HV and R-IGD;
- A web-based platform that allows scalability. The framework can run in the cloud computing, supporting optimization problems with large instances and number of objectives. It is possible to see the executions from anywhere.

An open-source implementation of *Nautilus Framework* is available[1]. The application of optimization algorithms generate solutions that have been proved to increase the effectiveness and efficiency of many software engineering tasks. *Nautilus Framework* contributes to fulfill new demands required by the nowadays 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.

[1]https://github.com/nautilus-framework

Acknowledgments

This work is supported by CAPES and CNPq, grants: 307762/2015-7 and 473899/2013-2.

REFERENCES

1. M. Harman, "The role of artificial intelligence in software engineering," in *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*, June 2012, pp. 1–6.
2. L. Ford, "Artificial intelligence and software engineering: a tutorial introduction to their relationship," *Artificial Intelligence Review*, vol. 1, no. 4, pp. 255–273, Dec 1987.
3. A. Ghannem, G. El Boussaidi, and M. Kessentini, "On the use of design defect examples to detect model refactoring opportunities," *Software Quality Journal*, vol. 24, no. 4, pp. 947–965, 2016.
4. B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, "On the use of machine learning and search-based software engineering for ill-defined fitness function: a case study on software refactoring," in *International Symposium on Search Based Software Engineering*. Springer, Cham, 2014, pp. 31–45.
5. M. Kessentini, A. Ouni, P. Langer, M. Wimmer, and S. Bechikh, "Search-based metamodel matching with structural and syntactic measures," *Journal of Systems and Software*, vol. 97, pp. 1–14, 2014.
6. M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, pp. 833–839, Dec. 2001.
7. C. L. B. Maia, T. F. do Nascimento, F. G. de Freitas, and J. T. de Souza, "An evolutionary optimization approach to software test case allocation," in *International Conference on Computational Intelligence and Information Technology*. Springer, Berlin, Heidelberg, 2011, pp. 637–641.
8. H. L. Jakubovski Filho, T. N. Ferreira, and S. R. Vergilio, "Preference based multi-objective algorithms applied to the variability testing of software product lines," *Journal of Systems and Software*, vol. 151, pp. 194–209, 2019.
9. 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.
10. A. V. Tsyganov and O. I. Bulychov, "Implementing parallel metaheuristic optimization framework using metaprogramming and design patterns," in *Information Technology Applications in Industry*, ser. Applied Mechanics and Materials, vol. 263. Trans Tech Publications Ltd, 2 2013, pp. 1864–1873.

11. S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA - a platform and programming language independent interface for search algorithms," in *Proceedings of the 2nd International Conference on Evolutionary Multi-Criterion Optimization (EMO '03)*. Faro, Portugal: Springer, 2003, pp. 494–508.
12. J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760–771, Oct. 2011.
13. D. Hadka, "MOEA Framework: A free and open source Java framework for multiobjective optimization. user manual," http://www.moeaframework.org/, 2016, Accessed in 20th August 2019.
14. E. O. Scott and S. Luke, "ECJ at 20: toward a general metaheuristics toolkit," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19)*, 2019, pp. 1391–1398.
15. Y. Tian, R. Cheng, X. Zhang, and Y. Jin, "PlatEMO: A MATLAB platform for evolutionary multi-objective optimization," *IEEE Computational Intelligence Magazine*, vol. 12, no. 4, pp. 73–87, 2017.
16. V. Ojalehto and K. Miettinen, "Desdeo: An open framework for interactive multiobjective optimization," in *Multiple Criteria Decision Making and Aiding*. Springer, 2019, pp. 67–94.
17. F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.
18. B. Li, J. Li, K. Tang, and X. Yao, "Many-objective evolutionary algorithms: A survey," *ACM Computing Surveys*, vol. 48, no. 1, p. 13, Sep. 2015.
19. T. N. Ferreira, S. R. Vergilio, and M. Kessentini, "Many-objective search-based selection of software product line test products with Nautilus," in *The 24th International Systems and Software Product Line Conference (SPLC '20) - Demo Track*. Montréal, Canada: ACM, 2020.
20. 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 Computational Intelligence Magazine*, vol. 12, no. 2, pp. 34–45, May 2017.

Thiago do Nascimento Ferreira received the PhD degree in Computer Science from the Federal University of Paraná, in 2019. His main interest are bio-inspired computation, multi-objective optimization and preference-based optimization algorithms focused on Search-based Software Engineering (SBSE). Contact him at tnferreira@inf.ufpr.br.

Silvia Regina Vergilio is currently a professor of Software Engineering in the Computer Science Department of Federal University of Paraná (UFPR), Brasil, where she leads the Research Group on Software Engineering. She has involved in several projects and her research is mainly supported by CNPq (PQ Level 1D). Her research interests include software testing, software reliability, Software Product Lines (SPLs) and Search-based Software Engineering (SBSE). She serves as assistant editor of the Journal of Software Engineering: Research and Development, and acts as peer reviewer for diverse international journals. She serves on the Program Committee of many conferences related to Search-Based Software Engineering and software testing. Contact her at silvia@inf.ufpr.br.

Marouane Kessentini is a recipient of the prestigious 2018 President of Tunisia distinguished research award, the University distinguished teaching award, the University distinguished digital education award, the College of Engineering and Computer Science distinguished research award, 4 best paper awards, and his AI-based software refactoring invention, licensed and deployed by industrial partners, is selected as one of the Top 8 inventions at the University of Michigan for 2018 (including the three campuses), among over 500 inventions, by the UM Technology Transfer Office. He is currently a tenured associate professor and leading a research group on Software Engineering Intelligence. Prior to joining UM in 2013, He received his Ph.D. from the University of Montreal in Canada in 2012. He received several grants from both industry and federal agencies and published over 110 papers in top journals and conferences. He has several collaborations with industry on the use of computational search, machine learning and evolutionary algorithms to address software engineering and services computing problems. Contact him at marouane@umich.edu.