AN EMPIRICAL FRAMEWORK TO APPLYING UNIT TESTING IN OPERATIONAL RESEARCH TO IMPROVE MAINTAINABILITY

M. Vidoni[1], M.L. Cunico[1],andA. Vecchietti[1*]
[1]Institute of Design and Development, INGAR CONICET-UTN
Avellaneda 3657, Santa Fe, Argentina
{melinavidoni, laura-cunico, aldovec}@santafe-conicet.gov.ar

ABSTRACT

Operational Research (OR) models usually deal with uncertain, changing requirements. This leads to a continuous process of adapting and reworking the mathematical code. However, there are scarce mechanisms to control its quality. This is essential to Software Engineering (SE), as it enforces the use of *Unit Testing*: automatically running tests after any alterations, to assess specific parts of the code. This is done to discover where and how errors are happening, simplifying its correction while evaluating their possible ramifications. This article aims to define how these concepts can be adapted to them, how tests should be used to detect faults and to provide a workflow to use them while developing an OR model. It provides guidelines on what should be tested and what to expectof possible errorsand a process to use it.

Keywords: Operational Research, Software Engineering, Unit Testing, Decision Support Systems.

[*] Corresponding Author

1 INTRODUCTION

In the last decades, given the increased computational capacities of computers and software, Operational Research (OR) mathematical models are becoming more complex,with more difficult maintenance. The model design and support is an important issue to take into account since it is subjected to continuous changes in its requirements during its lifecycle [3]. It is essential tokeep models up to date[4],and for this purpose, it is crucial to incorporate techniques that can enhance a model's capability to adapt to changes while reducing the risk of introducing errors [10].

In this article, it is discussed the use of *Unit Testing* (UT), which is a technique of Software Engineering (SE) that can be adopted for ORmodels. Unit Testing assesses individual units of source code, to discover if they work as expected [17]. The idea behind UT is to write test cases along with the regular code, and automatically execute themeach time a new codeis writtenorexisting one is modified, with the purpose of discovering errors [7].

Adapting UT to OR brings several advantages.First, it helps to control and reduce the number of errors introduced after altering the code, as well as decreases the time and workrequired to locate those errors. Second, UThelps to reuse tested code by lessening the effort needed to adapt it and,finally, it can ensure that the model behaves according to the expectations[23,24].

Because many languages used to implement OR models -such as Octave, MatLab, Python, R,and others- already provide automated Unit Testing features[7], this research aims to define how these concepts can be adapted to OR models, how tests should be used to detect faults and to provide a workflow to use them while developing a model.

2 UNIT TESTING CONCEPTS

In SE, general*testing* is an investigation conducted to provide stakeholders with information about the quality of the product or service under test[15].There are at least four levels of testing, as summarized as columns in Figure 1[16]:

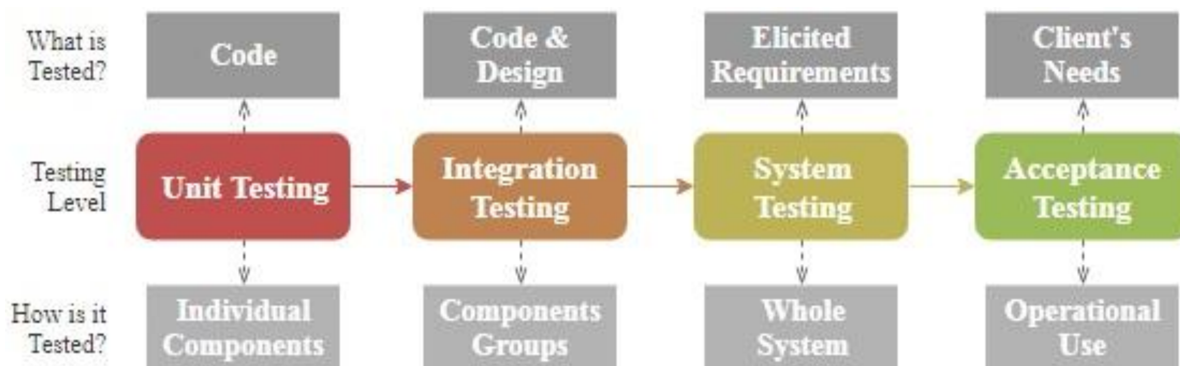

Figure 1.Levels of software testing, what and how it is tested.

Usually, a UT follows a predefined structure, regardless of the framework or language used. It is composed of two main parts:

- <u>Assertions</u>are statements where a predicate is always true at that point in code execution[18]. It works by sending specific known data to a function and comparing its result to an expected outcome. An outcome is not only positive –the function working correctly- but it canalso bean expected warning or error. In the latter, the assertion will be true if the error appeared[16].
- <u>Tests</u> group assertions that evaluate the same function with either different results or expected outcomes[18]. This way, it possible to run a single test to cover a wide range of aspects for the same target.

Testing is a fundamental activity of SE, asthe software will always need to be eventually tried andmonitored[15].Regarding its benefits,UT allows finding deeper, intrinsic errors than manual testing [9], improving the reusability and repeatability of the code.This builds a mechanism that ensures that no *compound errors* –i.e., those that do not break the code but conflict with requirements- are present [19], generating a predictable code base [16].

3 TESTING PROCESS IN OR

Traditional OR testing and UT are different concepts, yet complementary. In OR, testingis understood as checking that the obtained solution follows the logic of the modeled problem; for example, evaluating that if there is stock of a product, it has been manufactured in any previous period. However, UT searches for *compound errors*. This means that the written code must be coherent with what was modeled; this is to say that it actually solves the targeted situation, without considering the solution's optimality.

This article considers porting UT to OR, because SE has made several contributions to OR [13,14], with *Soft OR* methodologies being the most relevant [1], and since many mathematical languages widely used for OR models –such as MatLab, Octave, R, Python, and others- already provide UT functionalities [7].

Figure 2 shows a comparison between a traditional OR implementationand a Test-Driven one that applies UT.

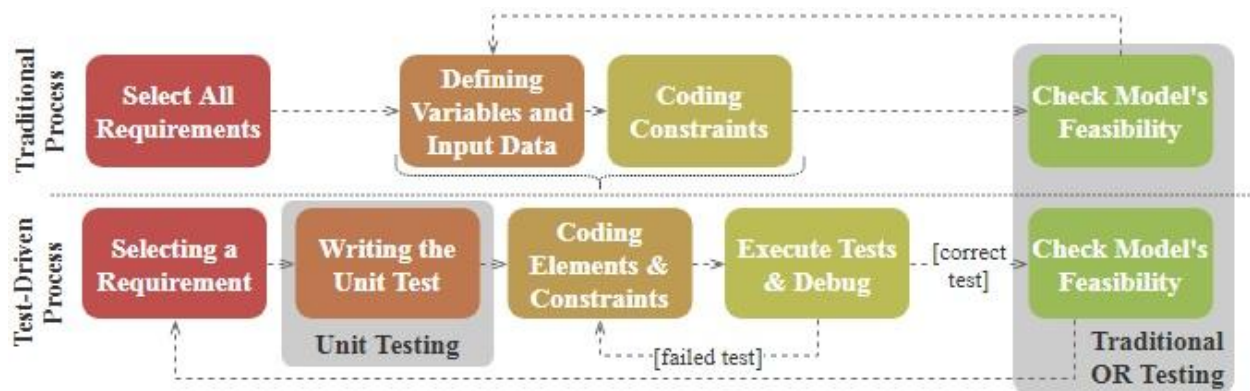

Figure 2.Traditional OR development, compared to Test-Driven

As a result, traditional OR testing is applied before or after running each model scenario; i.e., it is performed at *use time*, every time the model is run to find a particular solution. However, tests generated for UT are only used while writing new code, after correcting a specific error, when simplifying or improving existing code, or after altering it; this is to say, they are only executed at *development time*[15]. For example, while traditional OR testing checks that a product to be producedis tabulated in the input data, UT evaluates that the data is correctly read and parsed and that their type and dimensions match the expected ones.

This new, test-driven development in OR implies coding model functions one at a time, iteratively building an incremental, whole working model [22]. After selectinga functionality, tests should be written first. The element is thenwritten and refactored –i.e., improved- until the corresponding assertions pass.

Another important point is that tests need to be delivered along with the code [16]. This is crucial for models that are used daily or weekly as part of an organization process, as they are more vulnerable to changes and new requirements.

4 TESTS CASES IN OR

In OR, the *units* to be assessed through UT are input data, variables, constraints and objectives. Their tests are not exclusive, as they complement each other, helping to

discover where errors are located and how they are happening.

The following subsections detail how each of these units are tested, and what to expect from them. Here, code samples are written in *pseudo-code*; this is done to avoid limiting the proposal to a specific language;Section 4 will discuss how these are implemented in specific case studies. Thus, the code fragment (C1) showcases the pseudo-code parts of an assertion:

```
assert(type_of_assertion, tested_unit, comparison, expected_value)
```
(C1)

It is relevant to mention that the expected value is always known:

4.1 Testing Input Data

Input data is a group of known or predicted values that limit the problem, its variables and relationships, such as planning periods, raw material prices,products, demandforecasts, and so on [21]. The goal of testing input data is to discern if the read values are the same as the expected ones.

There are three sources for them: (a) embedded directly into the mathematical model, known as *hard-coded*, (b) imported from external sources, such as databases, spreadsheets, text files, and others, or (c) calculated in the model by using input data of sources (a) or (b).In any case, it can be a single number, or structured in vectors, matrices, tables or data-frames. More relevant tests are obtained when evaluating the latter.

Though testing *hard-coded* data can detect typographic and similar mistakes, it is more useful when depending on external sources. In this case, it helps to find importing errors, incorrect parsing, and encoding defects.The following is a non-exhaustive list of assessments, its goals, and example cases:

- Assertthe value of a specific element of a matrix, to determine if values are correctly read and if any parsing error is happening.For example, if reading from a CSV file, a failed assertion could mean that some values are using a comma, with a decimal number written as `3,15`instead of `3.15`, or a name including said character, leading towards a parsing error.
- Assert the type of a specific element to discover if data was adequately converted while reading. For example, if a matrix should contain integer values and the test detects it is read as a string, it may imply an incorrect use of the importing functions or another parsing error. Code (C2) shows a pseudo-code example.

```
assert(true, myMatrix[2,4], getType, Integer)
assert(false, myMatrix[2,4], getType, String)
```
(C2)

- Assert the dimensionsto discover if all values are read. For example, if a data-frame of expected (10,20) dimensions contains more columns, it may lead to an internal error; however, if a table has fewer rows than expected, it meansthat either the source is corrupted or the reading was halted.
- Assert if the external sources exist and if they are located in the specified location. Therefore, if it exists but it fails to read, the error may be caused by an incorrect connection, a corrupted file, or a misused reading function.

The third source is calculated input data. There should be one test for each specific calculus, including multiple asserts. Some possibilities for assertions are:

- Comparing to expect correct and incorrect values. For tabular data, these assertions can be done for the whole, or for individual cells. It is the most comprehensive assertion, as it evaluates if both the calculus structure (dimensions and operations order)and the values involvedare correct.

- Evaluating the size of the matrix. A failed calculus may result on a table of incorrect dimensions. By assessing the number of rows and columns, it ensures that the structure of the elements used is correct.
- Checking the internal data type. Many mathematical languages allow writing tabular where each cell can be of a different data type. These assertions should compare the value types (either individually, or by row/column, or completely), to discover if any parsing error is happening; these may be due to incorrect functions used on the calculus.

4.2 Testing Variables

*Decision variables*represent entities of the problem whose values are unknown until the mathematical model is solved[21]; examples are the number of equipment, available personnel, warehouses, and others.Two types of testsare recommended:

- Evaluate the variable initialization, to discover if it is correctly set up. This implies testing its type –positive integer, floating point, and others- and quantity. In some languages, this test may not be required, as it is checked on the compilation stage, previous to the solving.
- Gauge which values can be assumed by a variable. This is done in conjunction with the model's constraints and is discussed in the next subsection.

4.3 Testing Constraints

Constraints are logical propositions or symbolic algebraic relationships -set of equalities and inequalities- that delimit the search space and are responsible for linking input data and variables to influence the model classification[21].Constraints can also be clustered by *constraint groups*: subsetsthatrepresent a specific portion of the target situation. For example, a set of four equations –i.e.,individual constraints- that limit the purchase of raw materials; there, each equation represents provider selection, units to be purchased, purchase and transport cost.

As a result, constraints UT is the coreassessment that needs to be carried out. This process involves checking constraints incrementally. Hence, there should be three levels of tests; this is exemplified using an example of transport limitations constraints.

1. For individual constraints, as it ensures that there are no compound errors. This is to say that all constraints work adequately by themselves. Then, if one ofthese tests fail, the error search can be limited to a specific constraint.In the context of the example, this is a test that ensures that the maximum capacity of any truck is never exceeded.
2. For constraint groups. These tests allow moving towards an Integration Testing (see Figure 1), appraising how the constraints interact: it is possible for them to work adequately by themselves, but create a conflict when they are working together. Following the example, this test evaluates that a truck is assigned a load (through one constraint), only if it is chosen (using another constraint).
3. For increasing combinations of groups. As on (2), it allows checking different levels of the integration, until the whole model is appraised. The goal is to detect possible conflicts between groups and their interaction. In the example, this could be the case for merging transport and cost constraint groups, through a test that ensures that routes that are not chosen do not generate additional costs.

Evaluating constraints imply directly appraising the feasibility region (FR). Because the process is incremental, it can take manyshapes until it reaches the final form delimited by all the constraints.These are named as *temporal feasibility regions* (TFR), and all of themneed to be evaluated.

Thisis done by asserting if a point –a known value for the variables- belongs to each TFR or

not. As the TFR change, it is possible that a point that belongs to a previous simpler TFR would not belong to another more complex, or to the final FR. As this is not the only possible type of assertion, Figure 3 summarizes other options.

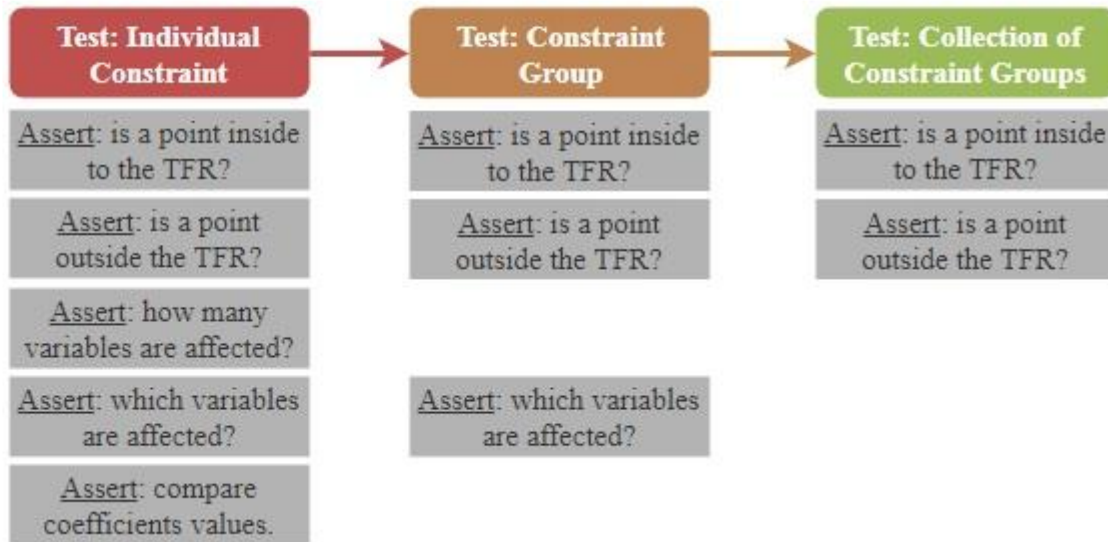

Figure 3.Types of assertions for each constraint test level.

It is possible to see that checking the TFRs is relevant at all levels, other assertions -such as checking the coefficient values- are only useful for individual constraints. Even more, depending on how the filestructure of a model is organized, it could be convenient to assert if the files exist on the specified location, to distinguish between corrupted and non-existent files.

Regarding expected values, it is worth noting that they must be on the central zone of the TFR, but also at the edgeto detect solutions sensitive to rounding.

4.3.1 TFR Evaluation Example

A running example will be used to deepen in the TFR assessment.

Suppose a model with two variables, $x_1 \geq 0$and $x_2 \geq 0$, with the constraints seen on equation (E1), where $eq1$ and $eq2$ represent a constraint group. In a rice packaging factory, the variables are the quantity of packaged containers of 1kilo (x_1), or half a kilo (x_2) of product. There, $eq1$ refers to the maximum packaging capacity, and $eq2$ establishes a relation between the packages generated, caused by a warehouse limitation; finally, $eq3$ corresponds to demand satisfaction. Thus, the first constraint group represents capacity, and the second one sales.

$$\begin{aligned} &eq1:\ 2 * x_1 + 3 * x_2 \leq 12 \\ &eq2:\ -x_1 + x_2 \leq 1 \\ &eq3:\ 3 * x_1 + 2 * x_2 \geq 12 \end{aligned} \qquad \text{(E1)}$$

The goal is to assess the valuesthat the variables can assume; this is done by manually selecting a known point and evaluating if it belongs to a TFR or not. Some example assertions for this case are:

I. Two assertions: (a) assert if the point $(x_1, x_2) = (6,1)$ belongs to the TFR defined by $eq3$ should be true (Figure 3-A).
II. Check the constraint group ($eq1$and $eq2$) by asserting that $(x_1, x_2) = (6,1)$ belongs to the TFR. This should be true (Figure 3-B).
III. Asserting if point $(x_1, x_2) = (4,2)$ belongs to the final FR (Fig. 3-C) should be false.

IV. Corroborating that point $(x_1, x_2) = (6,1)$ should be true for the final FR (Fig. 3-D).

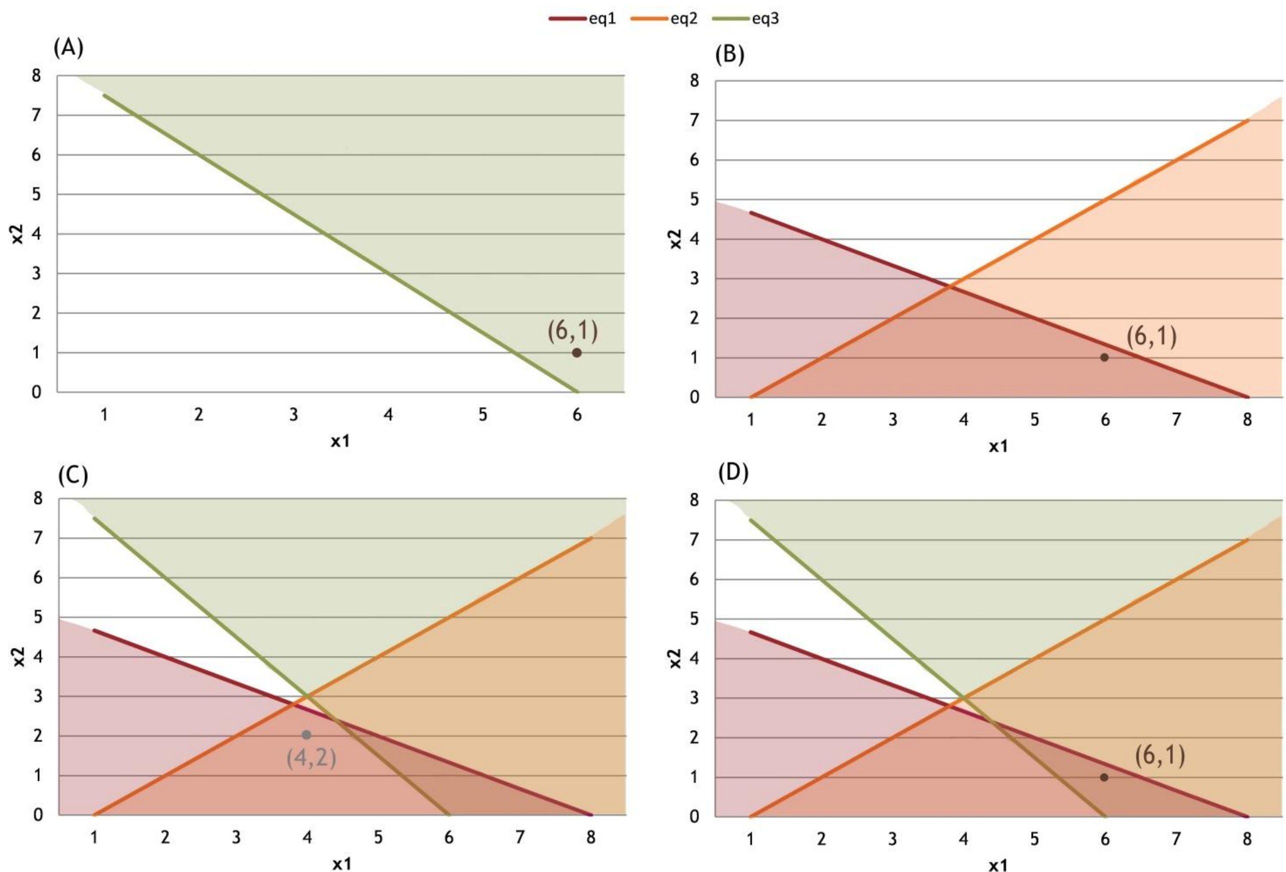

Figure 4.Visualizations of asserts I-IV on plots A-D, respectively.

In this case, stating that "an assertion failed" implies that it didn't behave as expected. For example, if assert (I) fails, then the point $(x_1, x_2) = (6,1)$ is outside the TFR, when it should be inside it (as pictured in Figure 3-A). The reverse happens if assertion (III) fails: in this case, the point $(x_1, x_2) = (4,2)$ belongs to the final FR, when it should be outside of it.

Then, if assert (I) and (IV) fail -they are not true- it is possible to assume that the error is being caused by a fault on $eq3$ (the demand is not satisfied), but not on the whole model.However, a deeper analysis depends on the result of assert (II):

- If (II) passes, this means that $eq3$ is incorrectly written, generating a different TFR. It could be the case where inequality ($\geq$) has been written as an equality ($=$); as a result, the fault in (IV) is caused only by $eq3$. In the running example, this means that the capacity is correctly modeled, but the demand satisfaction is too restricted.Therefore, the modeler needs to fix $eq3$ and run the tests again.
- However, if (II) also fails, it means that both the demand satisfaction and constraint capacity were incorrectly modeled, generating different TFR. In this case, additional assertions are needed to find which constraint of the capacity group is failing. As a result, fixing this error requires correcting both sets of equations.

This flow of thought contributes to discriminate faults from their symptoms, allowing an easier detection by discarding possibilities.The pseudo-code (C3) transcribes the assertions:

```
assert(true, eq3, includesPoint, [6,1])
assert(true, eq1 & eq2, includesPoint, [6,1])
assert(false, eq1 & eq2 & eq3, includesPoint, [4,2])
assert(true, eq1 & eq2 & eq3, includesPoint, [6,1])
```
(C3)

It is worth noting that this is a simplified example used to visualize what type of

corroborations mustbe done, and what possible results they could imply. This is not exhaustive,as more assertions –grouped on tests- should be written for a real case.

4.4 Testing Objectives

Objectives are the search direction used in the problem solution, and at least one objective functiondetermines it[21].Two points can be considered: (a) the values assumed by the variables used in the objective are limited by the constraints, and tested with them (b) evaluating the objective does not imply discovering the global optimum, because doing this requires solving the model.

Another possibility to test the objective functions is discovering whether the search direction (maximization or minimization) is indeed the expected one. For example, in a maximization problem, this means stopping the solving process at different iterations to check that the objective function is subsequently providing better results.

5 CONCLUSIONS

This paper presents guidelines for the creation and use of Unit Testing in OR. It focuses on what should be tested and what possible errors to expect,showing example tests to be performed. This presents a change of paradigm compared to traditional OR testing, as it aims to be applied at developing time, every time the source code is altered.

This is done to provide a tool for improving the quality of the code, giving the models the capacity to adapt to the changes. Even more, it simplifies code reuse by allowing salvaging working, tested code and using it again in another project. Testing can contribute to teamwork by prematurely detecting the occurrence of errors in parts developed by different modelers. However, though the simultaneous implementation of the tests and main code reduces the error detection times, it does not guarantee a fault-free code.

Spendingadditional time onUnit Tests in OR interventions may be a source of resistance to this proposal, due to the increased requirements in coding, which are linked tothe limited project's deadlines. The fact that adding UT alters the whole development stage of a model, and that it has a steep learning curve, also contributes to this situation. Additionally, even if some mathematical languages already provide automation tools for Unit Testing that can be used for this, others do not.

However, this proposal does not pretend to be exhaustive since it will primarily depend on the problem and the programming language used. Future lines of work are generating automated tools,refining the tests to be carried out, and evaluating its use in real-world interventions.

6 ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support for the work presented in this article to Universidad Tecnológica Nacional (UTN) through PID EIUTIFE0003974TC.

7 REFERENCES

[1] L. Fortuin and M. Zijlstra, "Operational research in practice: Consultancy in industry revisited," *European Journal of Operational Research*, vol. 120, no. 1, pp. 1-13, 2000, https://doi.org/10.1016/S0377-2217(98)00377-4.

[2] A.P. Wierzbicki, "Modelling as a way of organizing knowledge," *European Journal of Operational Research*, vol. 176, no. 1, pp. 610-635, 2007, https://doi.org/10.1016/j.ejor.2005.08.018.

[3] G. Büyüközkan and O. Feyzioğlu, "Group Decision Making to Better Respond Customer Needs in Software Development," *Computers & Industrial Engineering*, vol. 48, no. 2,

pp. 427-441, 2005, https://doi.org/10.1016/j.cie.2005.01.007.

[4] D. Huizinga and A. Kolawa, *Automated Defect Prevention: Best Practices in Software Management*, 1st ed. New Jersey, USA: Wiley-IEEE Computer Society, 2007.

[5] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," in *25th International Symposium on Software Reliability Engineering*, Naples, Italy, 2014, pp. 201-211, https://doi.org/10.1109/ISSRE.2014.11.

[6] L. Williams, E.M. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice," in *14th International Symposium on Software Reliability Engineering*, Denver, USA, 2003, pp. 34-45, https://doi.org/10.1109/ISSRE.2003.1251029.

[7] B. Turhan, L. Layman, M. Diep, F. Shull, and H. Erdogmus, "How Effective is Test Driven Development," in *Making Software*, 1st ed., A. Oram and G. Wilson, Eds. USA: O'Reilly Media, 2010, ch. 12, pp. 207-2019.

[8] G.J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Canada: John Wiley & Sons Inc., 2012.

[9] P. Bourque and R.E. Fairley, *Guide to the Software Engineering Body of Knowledge*, 30th ed.: IEEE Computer Society, 2014, http://www.swebok.org/.

[10] E. Mera, P. Lopez-García, and M. Hermenegildo, "Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework," in *International Conference on Logic Programming*, vol. LNCS 5649, Pasadena, USA, 2009, pp. 281-295, https://doi.org/10.1007/978-3-642-02846-5_25.

[11] L. Gren and V. Antinyan, "On the Relation Between Unit Testing and Code Quality," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Vienna, Austria, 2017, pp. 52-56, https://doi.org/10.1109/SEAA.2017.36.

[12] D.M. Rafi, K.R.K. Moses, K. Petersen, and M.V. Mäntylä, "Benefits and limitations of automated software testing: systematic literature review and practitioner survey," in *7th International Workshop on Automation of Software Test*, vol. 1, Zurich, Switzerland, 2012, pp. 36-42.

[13] J. Mingers and L. White, "A review of the recent contribution of systems thinking to operational research and management science," *European Journal of Operational Research*, vol. 207, no. 3, pp. 1147-1161, 2010, https://doi.org/10.1016/j.ejor.2009.12.019.

[14] R.J. Ormerod, "The transformation competence perspective," *Journal of the Operational Research Society*, vol. 59, no. 11, pp. 1435–1448, 2008, https://doi.org/10.1057/palgrave.jors.2602482.

[15] J.C. Ranyard, R. Fildes, and T.-I. Hu, "Reassessing the scope of OR practice: The Influences of Problem Structuring Methods and the Analytics Movement," *European Journal of Operational Research*, vol. 245, no. 1, pp. 1-13, 2015, https://doi.org/10.1016/j.ejor.2015.01.058.

[16] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43-50, 2005, https://doi.org/10.1109/MC.2005.314.

[17] D.G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*, 4th ed. Switzerland: Springer International Publishing, 2016, vol. 228, DOI: 10.1007/978-3-319-18842-3.