

# Towards Modelling Acceptance Tests as a Support for Software Measurement

Alexandra Lapointe-Boisvert  
Université du Québec à Montréal  
Montréal, Canada  
lapointe-boisvert.alexandra@uqam.ca

Sébastien Mosser  
Université du Québec à Montréal  
Montréal, Canada  
mosser.sebastien@uqam.ca

Sylvie Trudel  
Université du Québec à Montréal  
Montréal, Canada  
trudel.s@uqam.ca

**Abstract**—The DevOps paradigm emphasizes the need for a measurable feedback loop, starting from requirements and going as far as deployment in an automated way. In this context, a modelling challenge is to leverage the existing requirement engineering approaches to support measurements. Unfortunately, measurement methods are slow and costly by definition, preventing precisely measured requirements from being used in the DevOps loop. As a result, developers have to deal with grossly estimated elements, *e.g.*, using story points promoted by agile methods. Thus, it is not possible to provide better support for the development team. We envision taking advantage of the artifacts that already exist in a DevOps context to provide better support for requirements measurement, making it available in an automated context such as the DevOps one. This paper focuses on the automated analysis of acceptance tests (*e.g.*, expressed using the Gherkin language) to support functional measurement automation in a DevOps context. This proposition is illustrated by a scenario coming from an industrial partner, supporting the identification of four research challenges to be tackled.

## I. INTRODUCTION

Many organizations shift from waterfall to Agile and now DevOps culture to deliver software rapidly and continuously, be more adaptive to change, and achieve their performance goals. This is done by adopting leaner processes, disabling non-added value activities and leveraging automation tools to speed and stabilize software delivery through the entire product life cycle [1]. With DevOps accelerating the pace of digital development, all development actors must adapt. Instead of relying on holistic specifications, development teams break down products into smaller pieces of functional software to release frequently and on cadence. DevOps practitioners are continuously improving *Continuous Integration* (CI) and *Continuous Deployment* (CD) pipelines to integrate and deploy more efficiently. Quality analysts are automating test executions to ensure the quality of the software being shipped at every iteration. Security is integrated into every phase of the software development life-cycle. By its very essence, DevOps is about providing a way to continuously improve the development process in an endless feedback loop. *Thus, measurement becomes critical, as one cannot improve what cannot be measured.*

From a state-of-practice point of view, user requirements are mainly written into the form of *user stories* and functional tests are written based on the acceptance criteria of the *user story*. Tests tend to be written simultaneously, and user stories

are required and act as functional requirements that prevent duplication of requirements writing. Eventually, the DevOps paradigm reaches a paradox here, by emphasizing the need to improve while still relying on clumsy measurements to support such improvement at the requirements level. More precise way of measuring software exists (*e.g.*, the COSMIC method [2]), with demonstrated benefits in terms of planning and project implementation [3]. Unfortunately, the effort necessary to measure a given piece of software according to these methods is tremendous, making it close to impossible to integrate such approaches in a DevOps context.

Any research effort in this direction should take into consideration the state of practice: while relying on informal artifacts as inputs, there is a need for better measurements to support the DevOps feedback loop. As a consequence, the global research question to be addressed is the following:

**RQ:** “*To what extent existing artifacts developed in a DevOps context can be modelled to support a better (faster and more precise) estimation at the requirements level?*”

In the remainder of this paper, we envision how to address this question by focusing on modelling *acceptance tests* as relevant DevOps artifacts to support COSMIC measurements allowing more precise measurements to be obtained in a faster way. SEC. II describe the research effort in this context, and SEC. III provides an in-depth description of an illustrative scenario with the associated artifacts available. We describe in SEC. IV our vision to address the research question by refining it into modelling challenges to be tamed. Finally, section SEC. V concludes this paper by summarizing the challenges and the benefits of the approach for requirement engineers working in a DevOps context.

## II. RELATED WORK

There is a *de facto* convergence existing between Requirements Engineering (RE) and DevOps approaches. It started with the study of agile methods (*e.g.*, [4], [5], [6]) and naturally evolve into taking into account the DevOps paradigm at a broader level [7]. The objective of this section is to picture how DevOps and RE research efforts evolved, while emphasizing the lack of dedicated modelling approach in this context.

Agile methods are considered successful by the industry and are widely used in DevOps ecosystems. However, one of the

main remaining challenges here is the effort estimation, as it relies on experts' opinions or judgements [8]. The concrete way of estimating varies, as there are many different agile estimation techniques. One common technique is to use poker planning to evaluate a given requirement's relative complexity in *story points* [9]. However, there are many drawbacks to this approach. Estimating in story points has been shown to consume much time, for very subjective results [8]. Secondly, story points are often misunderstood between the different actors of development. Another wrong usage of story points is using them to compare team performance or evaluate individual performance. Moreover, story points do not provide any size metrics [10] that can be used to improve a DevOps loop.

At the other side of the spectrum, functional measurement methods are available. They promote measurement as essential for managers to make better decisions: it is nearly impossible to properly estimate effort without having a minimum understanding of the product size to develop [11]. The ISO/IEC 19761:2011 normative document [2] describes the COSMIC functional size measurement method to measure software size as COSMIC function points. Several studies have proposed to measure functional size with COSMIC from different input artifacts. Marín *et al.* [12] have conducted a survey in 2008 identifying existing measurement procedures that have been proposed for applying the COSMIC measurement method. Among the eleven methods inventoried, five measured UML models, one measured xUML specifications, one measured RRRT models, one measured MERODE models, two measured OO-Method models and one measured *i\** models as a primary artifact. They observed that seven out of eleven methods did not specify all the functional requirements and only allowed to estimate the functional size.

Zhu *et al.* [13] identified four challenges with automation of functional measurement: entity identification, entity disambiguation, entity-relationship extraction and event extraction, which are almost all done manually now. Several approaches are defined to measure automatically existing code [14], [15]. To support automation, alternative methods relying on model-driven engineering also propose to measure by leveraging conceptual models [16], [17]. Finally, Ochodek, Koczyńska and Staron [18] proposed a deep learning model to approximate COSMIC size from use cases automatically. All the approaches covered in this section suffer from either or both of the following flaws when considered in a DevOps context: (i) they are manual/time-consuming, or (ii) they rely on artifacts (e.g., UML use cases) that are not used by the state of practice in DevOps.

Focusing on such state-of-practice, the most common strategies in agile testing are *Test-Driven Development* (TDD), *Acceptance-Test-Driven Development* (ATDD), and *Behavior-Driven Development* (BDD) [19]. Acceptance tests are expressed as code (e.g., using the Gherkin language [20]), creating an executable bridge between the user stories they are validating the implemented product. Solís and Wang advocate that ATDD is widely adopted in the industry because it im-

proves software quality and productivity [21]. Longo showed that developers would use acceptance testing for specifying software requirements over natural language to improve the quality of requirements [22]. Fischbach *et al.* [23] studied 961 user stories from the industry and observed that, even if not systematic (e.g., "up to 51%"), ATDD is used on a regular basis by the industry.

To conclude this related work review, we as software engineering and modelling researchers are facing a challenge. The DevOps state-of-practice requires measurements to support its improvement feedback loop. Unfortunately, at the requirements levels, there is no way to provide proper estimations of the pieces of software to be developed, as the existing methods cannot be used in such a context: they would consume too much time, or would rely on models/artifacts that are classically missing in DevOps ecosystems. However, even if it is not possible to consider acceptance tests as actionable models, such testing approaches provide a trade-off to address this challenge: first they have the obvious advantage of existing in our context, and secondly, as such tests are expressed as code, they provide a solid ground to automate the extraction of relevant models from their implementation.

### III. SCENARIO FROM THE STATE-OF-PRACTICE

We consider here a scenario used to illustrate the artifacts available to support estimation in a DevOps context, as close as possible to the requirements level. We consider here a cross-functional team that includes a *product owner*, *software developers* to implement the product, and *operational engineers* to support its deployment. In a product-centric approach, the development team produces increments of the software product in iterations (i.e., *dev*), ultimately deployed in an operational way (i.e., *ops*). Since budget and time are fixed, the team is only required to evaluate and plan the scope produced over the next iteration.

Prior to the development iteration, the team meets with the customer or the product owner in order to define a functional increment of the product [24]. The result of this discussion is captured in a *User Story*, expressed as a triple (*role*, *feature*, *reason*). The story is classically written in natural language as a sentence, using the Connextra format<sup>1</sup> (FIG. 1). The story is then stored using a ticket management system such as Jira.

*As a level designer I would like to create custom grids in order to increase my level of challenge*

Fig. 1. Example of user story for a video game

When the story is groomed to be included in the development sprint, the story needs to be estimated (classically using *story points*). The team classically estimates the stories according to two dimensions: (i) business value, and (ii) technical risk. As shown in the previous section, these estimations are used in a *good enough* way, meaning that they are rough and not precise, but at least exist. The estimation is also stored

<sup>1</sup>"As a *role*, I want *feature* so that *reason*".

```

Feature: Level editor

Scenario: Access to the level editor.
Given I am on the main menu of the program
When I select the menu option "Level editor"
Then I am in the "Level editor" menu

```

Listing 1. Example of an acceptance scenario expressed with Gherkin

in the ticket management system as metadata. When a user story is adequate (e.g., it is independent between each other, negotiable, valuable, estimable, small and testable [25]), it is considered as *ready*.

When a story is ready, an essential corollary notion in the context of DevOps is the *Definition of Done* (DoD). With a precise DoD, it is possible to transfer a story from the “*dev*” space to the “*ops*” one as soon as possible, enacting a continuous deployment feedback loop. To support this, the development team starts writing acceptance tests from the story’s acceptance criteria. The success of the acceptance tests for this very feature then implements the DoD. During the development stage, the development team develops the product increment based on the acceptance tests’ specifications until the DoD is reached, i.e., successfully executing this very feature’s acceptance tests.

In this context, many organizations have adopted a structured language, such as Gherkin, as their preferred method of documenting test scenarios. The advantage of such an approach is that it minimizes the potential of ambiguities instead of requirements in natural language and allows the scenarios to be parsed easily, generally to automate the test execution, without compromising readability for non-technical stakeholders. We describe in LST. 1 an example of an acceptance scenario described using the Gherkin language. It allows one to define scenarios, as a succession of steps, according to a *given-when-then* format. A scenario describes (i) the context of the scenario using steps starting with *given*, (ii) the trigger of the feature with steps starting with *when*, and finally a sequence of assertions (i.e., using *then* steps) used to assess the scenario in an automated way. Acceptance tests are usually written alongside the *user stories* directly in Jira through test plugins (e.g., XRay, Zephyr) or specialized BDD testing tools (e.g., Cucumber, jBehave, RSpec).

The acceptance scenarios modelled using Gherkin are only the visible part of the iceberg. Using frameworks such as Cucumber, a Gherkin scenario is mapped to executable test code using regular expressions. This executable code is classically implemented using a unit test framework, as described in LST. 2. The execution engine takes as input a Gherkin file, matches each step of the scenario with the associated function, and executes it.

A company that is not mature enough would now stop the requirement engineering process and then move to the development, validation and deployment phases. However, a company more mature concerning its product development can start a measurement process to have a better estimation

```

@given('I am on the main menu of the program')
def step_impl(context):
    context.application.state =
        _main_menu(context.application)

@when('I select the menu option "{}"')
def step_impl(context, option_label):
    menu = context.application.state
    menu.activate_option(option_label)

@then('I am in the "{}" menu')
def step_impl(context, menu_title):
    menu = context.application.state
    assert isinstance(menu, menus.Menu)
    assert menu.title == menu_title

```

Listing 2. Implementing the scenario with unit tests (in Python)

of the size of the software to be implemented. The more precise a measurement is, the better it is possible to size the development iteration and anticipate that the features committed to being developed will be pushed to production at the end of the development iteration.

For this example, we consider a measurement process implemented according to the COSMIC method. In a nutshell, the idea of COSMIC is to measure the number of data movements inside a given piece of software, identified as *function points*. An expert will carefully evaluate the software’s source code and requirements, identifying which part of the code is related to which feature. Then, the method is applied to measure function points as data movements, e.g., data entering into the feature, exiting the feature, being stored or read from persistent storage. Each movement worth one function point, and the size of the feature is defined as the sum of all the function points involved. The challenge here is that this process is known to be time-consuming, as it is human-based. Moreover, it requires a deep understanding of the software to measure by the COSMIC expert. Consequently, it is not possible to include a measurement step inside each development iteration, preventing the usage of such methods in a DevOps context.

As one can notice, models are absent in this scenario. The objective of DevOps software development being to deliver value, classical models are seen as a loss of time in the DevOps state-of-practice, and existing industrial methods needs to be deeply adapted to consider classical modelling as a first-class citizen [26]. We describe in the next section our vision with respect to this situation.

#### IV. VISION, PROPOSITION & CHALLENGES

If models are absent from the state-of-practice, they are still required from an engineering point of view, to provide a formal ground to automation (in our case, supporting the measurement process).

According to the iterative and feature-driven approach emphasized by the DevOps paradigm, we define a product  $p \in \mathcal{P}$

```

def gcd(a: int, b: int):
    if a < b:
        a, b = b, a # Swapping a and b
    while b != 0:
        a, b = b, a % b # Computing the GCD
    return a

```

Listing 3. Greatest Common Divisor (GCD) algorithm in Python

as the composition ( $\oplus$ ) of products increments  $\delta_i \in \Delta$  (i.e., features) to the empty product ( $\emptyset \in \mathcal{P}$ ) [27].

$$p = \emptyset \oplus \delta_1 \oplus \dots \oplus \delta_i$$

As described in the previous section, we define a product increment  $\delta = (u, \{s_1, \dots, s_n\}) \in \Delta$  as a user story  $u$ , and a set of acceptance scenarios  $s_i \in \mathcal{S}$ . Classical (and manual) measurement methods would take  $u$  as input, but we envision here an automated measurement process that takes  $\delta$  as input. The measurement of a given increment is then defined as the composition (e.g., a sum) of the code's measurements associated with each acceptance scenario involved in the increment.

$$\begin{aligned}
 \text{measure} : \quad & \Delta \rightarrow \mathbb{N} \\
 (u, \{s_1, \dots, s_n\}) \mapsto & \sum_{i=1}^n \text{measure}(s_i)
 \end{aligned}$$

As a consequence, we need now a way to properly model each acceptance scenario to support their measurements.

#### A. Using Control-Flow as Modelling Foundation

Our proposition here is to investigate how compilation techniques can be used in this context. As acceptance scenarios are written as code, we envision an approach relying on control-flow extraction to automatically extract from the source code the *Control-Flow Graph* of the code involved in the acceptance scenario. A CFG is a classical data structure used in the compilation domain to represent a given program's execution flow, supporting static analysis (e.g., test generation [28]). We describe in LST. 3 an implementation of the greatest common divisor using the Euclidian's algorithm, and in FIG. 2 the associated CFG that can be automatically extracted from the source (e.g., using a function named *cfg*). CFG-based models are interesting from a modelling point of view, as (i) they can be easily extracted from source code, and (ii) they can be manipulated by large families of algorithms. Considering software measurement in particular, CFGs are helpful as they model the way instructions are modifying the processed data (which is precisely what software measurement captures).

As there is no reference consensus about how to model a CFG, we consider here a classical graph-based modelling. We define a CFG as a typical attributed graph  $G = (V, E)$  where  $V = \{v_1, \dots, v_i\}$  is a set of vertices and  $E = \{e_1, \dots, e_j\} \subseteq V \times V$  a set of edges. Vertices and nodes might contains attributes (e.g., labels for conditions, kind of instructions). The modelling space is built as a commutative

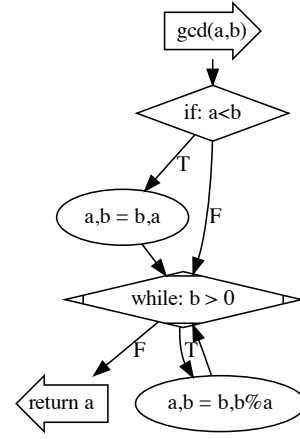


Fig. 2.  $g_{gcd}$ : Control-Flow Graph automatically extracted from LST. 3

monoid  $CFG = (G, \cup, \emptyset)$ , where  $\cup$  is the classical graph union operator<sup>2</sup> and  $\emptyset$  the empty graph.

To support the measurement process, we define three helper functions: (i) *calls* to extract the instruction calling the business logic used inside a scenario, (ii) *slice* to extract the CFG of the sub-part of the product covered by a given acceptance scenario, and (iii) *size* to measure a given CFG (e.g., computing COSMIC function points). Considering the previous example, if an acceptance scenario  $s$  contains a call to  $gcd(a, b)$ , then the sliced CFG will contain  $g_{gcd}$  (FIG. 2), added to the CFGs extracted from the other instructions involved in  $s$  using the  $\cup$  operator. Then, any measurement method can be used to *size* the resulting CFG. For example, using the COSMIC method, the data exchange between elements involved in the CFG will be used as main measurement points.

$$\text{calls} : \mathcal{S} \rightarrow \text{Instruction}^*$$

$$\text{slice} : \mathcal{S} \rightarrow \text{CFG}$$

$$s \mapsto \{g \mid \text{Let } \text{calls}(s) = \{i_1, \dots, i_n\}, \bigcup_{j=1}^n \text{cfg}(i_j)\}$$

To provide a sound model, we define a *measure* function as the composition of the previously defined helpers.

$$\begin{aligned}
 \text{measure} : \quad & \Delta \rightarrow \mathbb{N} \\
 (u, \{s_1, \dots, s_n\}) \mapsto & \sum_{i=1}^n \text{size}(\text{slice}(s_i))
 \end{aligned}$$

#### B. Research Challenges

The previous section sketches the foundations one can use to properly model product increments and the associated acceptance scenarios in the context of software measurement. Our proposition has the advantage of giving a precise value of the software size much earlier in the software development life cycle for better support of planning and decision

<sup>2</sup>Let  $g_1 = (V_1, E_1)$  and  $g_2 = (V_2, E_2)$ ,  $g_1 \cup g_2 = (V_1 \cup V_2, E_1 \cup E_2)$ . The operator  $\cup$  is by nature commutative and associative, as it relies on set unions. The empty graph  $\emptyset = (\emptyset, \emptyset)$  is its natural identity element.

making. However, finding a proper way to model existing artifacts triggers research challenges to be addressed in order to properly achieve this vision. To refine our initial research question, focusing on acceptance tests and our envisioned modelling paradigm, we have identified the following four research challenges that need to be tamed:

- $C_1$  *Modelling polyglot products.* Nowadays, products are rarely written using a single language. It triggers a challenge to define the *slice* function used to project a CFG from a given acceptance scenario. This projection can be made statically or by dynamically executing the tests. In addition to the engineering challenge of supporting multiple languages, the research challenge relies here on the analysis’s uncertainty. It is easy to extract a CFG from a program written in a statically typed language, but dynamic languages make the job much harder [29]. As a consequence, the CFG model extraction implemented by the *slice* function must include uncertainty mechanisms that will be propagated to the measurement obtained by the *size* function, as well as its composition for multiple tests at the *measure* function level.
- $C_2$  *CFG extrapolation of unwritten code.* It is classical in compiler engineering to extract control flow from a program representation. However, the challenge here is to work at the early phases of development, where the feature’s code is not implemented yet. In these conditions, it is not possible to perform a classical extraction. Instead, the key point is to leverage inference techniques used to extrapolate characteristics of the “code to be written” based on the code previously written for this product or a similar one in the company portfolio. Similar techniques were successfully applied in AAA video game development to identify the probability of code contributions to contain functional bugs [30]. Providing such model inference mechanism is critical, as well as identifying the limitations of the existing approaches.
- $C_3$  *Input data quality measurement.* Since the quality of the measurement is dependent on the quality of the acceptance tests, potential defects such as ambiguities or inconsistencies impact the quality of measurement. The input artifact must have enough semantic formalization to allow completeness of the functional requirements. The challenge here is to provide an automated evaluation method for the acceptance scenarios and consider this evaluation to characterize the uncertainty of the measure at the model level.
- $C_4$  *Adoption potential.* In order to adhere to a new paradigm, an organization must see its benefits. To better meet business demands and customers’ needs, increasing their productivity or decreasing response time to change are good incentives for an organization to revisit and change its processes. Leveraging acceptance tests to support functional measurement would reduce the effort in estimation and stem duplication effort of requirements writing, improve quality of requirements, and accelerate

delivery. The challenge here is to identify a reference benchmark to empirically demonstrate the company’s benefits when such measurements are available to improve the DevOps feedback loop.

### C. Methodology and Work in progress

The work described in this paper is done in close collaboration with an industrial partner, which provides unlimited access to the DevOps teams that work on several products. To date, the research effort was focused on the identification of the artifacts available in a DevOps context, following a comprehensive state-of-practice analysis started in 2019 in an M.Eng. thesis. These results were then confronted to the DevOps ecosystem implemented in the context of the industrial partner. This work leads to the definition of the scenario described in SEC. III. Based on this scenario, and an analysis of the state-of-the-art approaches described in the literature (SEC. II), we have identified the four challenges  $C_i$  previously described. We are now focusing our efforts on addressing (and hopefully tame) these challenges.

To achieve such a goal, we plan to leverage our industrial collaboration as a medium-scale case study by first charting the different teams’ DevOps practices at a fine-grained level. These practices will help us target the languages that will cover the most significant part of the product ( $C_1$ ) while limiting the engineering effort. As the products developed by the company can be considered as legacy, we have access to an extensive history of code modifications. These histories will be used to feed the CFG inference learning algorithms ( $C_2$ ) and validation benchmarks (we have access to the team that developed these features to qualitatively evaluate if an equivalence identified by the algorithms is compatible with the team domain knowledge). Considering that the company shifted to a DevOps context one product at a time, it gives us access to teams with different maturity levels. Thus, the evaluation of the maturity of the acceptance scenarios ( $C_3$ ) will be supported at two levels: (i) by comparing the scenarios written by different teams with a different level of expertise, and (ii) by comparing scenarios written in the past with more recent ones inside the same team. Finally, we will use this case study to identify the benefits measured at quantitative and qualitative levels for the industrial partner and reproduce the obtained result in other contexts ( $C_4$ ). To achieve such a goal, we will rely on an industrial-academic research chair dedicated to DevOps that accelerate the collaboration between universities and industries (e.g., empirical case study, qualitative studies).

## V. CONCLUSIONS

In this paper, we identified one of the biggest challenges for functional software measurement in the context of a DevOps ecosystem. Despite all of their advantages for the DevOps team, functional measurements cannot be included in a DevOps loop. In this context, our proposition is to identify which artifacts developed in a DevOps ecosystem can be leveraged to provide ways to automate such measurement. To date, we focused on acceptance scenarios defined using

the Gherkin language as the entry point for an approach that will support a better estimation of the functional size to be developed, starting as early as possible, and as close as possible to the requirements. This approach triggers four research challenges yet to be addressed, and we proposed a methodology to focus our research effort to tame these challenges. The challenges cover a wide spectrum of software engineering research, from compilation ( $C_1$ ) to code inference ( $C_2$ ), data quality ( $C_3$ ) and stakeholder adoption measurement ( $C_4$ ). Combining these four challenges into a comprehensive approach will provide an elegant and efficient way to reconcile requirements engineering and DevOps through a functional measurement point of view.

## REFERENCES

- [1] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*, 1st ed. Portland, OR, USA: IT Revolution Press, 2016.
- [2] ISO/IEC 19761-2, “Software engineering – COSMIC: a functional size measurement method,” International Organization for Standardization, Geneva, CH, Standard ISO/IEC TR 19761-2:2011, 2011. [Online]. Available: <https://www.iso.org/standard/54849.html>
- [3] J.-F. Dumas-Monette and S. Trudel, “Requirements engineering quality revealed through functional size measurement: an empirical study in an agile context,” in *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. IEEE, 2014, pp. 222–232.
- [4] S. Mosser and J.-M. Bruel, “Reconciling Requirements and Continuous Integration in an Agile Context,” in *International Requirements Engineering Conference*, ser. RE, Aug. 2018, Tutorial.
- [5] F. Dalpiaz and S. Brinkkemper, “Agile Requirements Engineering: from User Stories to Software Architectures,” in *International Requirements Engineering Conference*, ser. RE, Sep. 2021, Tutorial.
- [6] F. Dalpiaz, V. D. Schalk, B. Ivor, A. Sjaak, F. Başak, and G. Lucassen, “Detecting terminological ambiguity in user stories: tool and experimentation,” *Information and Software Technology*, vol. 110, pp. 3–16, 2019.
- [7] J.-M. Bruel and S. Mosser, “Requirements Engineering in the DevOps Era,” in *International Requirements Engineering Conference*, ser. RE, Sep. 2021, Tutorial.
- [8] P. Sudarmaningtyas and R. B. Mohamed, “Extended Planning Poker: A Proposed Model,” in *2020 7th International Conference on Information Technology, Computer, and Electrical Engineering (ICITACEE)*, 2020, pp. 179–184.
- [9] M. Usman, E. Mendes, F. Weidt, and R. Britto, “Effort estimation in agile software development: a systematic literature review,” in *Proceedings of the 10th international conference on predictive models in software engineering*, 2014, pp. 82–91.
- [10] C. Commeyne, A. Abran, and R. Djuab, “Effort estimation with story points and cosmic function points—an industry case study,” *Software Measurement News*, vol. 21, no. 1, pp. 25–36, 2016.
- [11] F. G. Wilkie, I. R. McChesney, P. Morrow, C. Tuxworth, and N. Lester, “The value of software sizing,” *Information and Software Technology*, vol. 53, no. 11, pp. 1236–1249, 2011.
- [12] B. Marín, G. Giachetti, and O. Pastor, “Measurement of functional size in conceptual models: A survey of measurement procedures based on COSMIC,” in *Software Process and Product Measurement*. Springer, 2008, pp. 170–183.
- [13] J. Zhu, S. Huang, Y. Shi, M. Chen, J. Liu, and E. Liu, “Survey on Methods for Automated Measurement of the Software Scale,” *International Journal of Performability Engineering*, vol. 16, no. 2, 2020.
- [14] A. Sahab and S. Trudel, “COSMIC Functional Size Automation of Java Web Applications Using the Spring MVC Framework,” in *Joint Proceedings of the 30th International Workshop on Software Measurement and the 15th International Conference on Software*
- [15] H. Soubra, Y. Abufrikha, and A. Abran, “Towards Universal COSMIC Size Measurement Automation,” in *International Workshop on Software Measurement – IWSM 2020*. Mexico: CEUR-WS, 2020.
- [16] B. M. Marín Campusano, “Functional size measurement and model verification for software model-driven developments: A cosmic-based approach,” Ph.D. dissertation, Universitat Politècnica de València, 2011.
- [17] S. Abrahão, J. Gómez, and E. Insfran, “Validating a size measure for effort estimation in model-driven Web development,” *Information Sciences*, vol. 180, no. 20, pp. 3932–3954, 2010.
- [18] M. Ochodek, S. Kopczyńska, and M. Staron, “Deep learning model for end-to-end approximation of COSMIC functional size based on use-case names,” *Information and Software Technology*, vol. 123, p. 106310, 2020.
- [19] S. O. Barraood, H. Mohd, and F. Baharom, “A Comparison Study of Software Testing Activities in Agile Methods,” in *Knowledge Management International Conference (KMICE) 2021*, Malaysia, 02 2021.
- [20] E. C. dos Santos and P. Vilain, “Automated acceptance tests as software requirements: An experiment to compare the applicability of fit tables and gherkin language,” in *International Conference on Agile Software Development*. Springer, 2018, pp. 104–119.
- [21] C. Solis and X. Wang, “A study of the characteristics of behaviour driven development,” in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2011, pp. 383–387.
- [22] D. H. Longo, P. Vilain, and L. P. da Silva, “Impacts of Data Uniformity in the Reuse of Acceptance Test Glue Code,” in *SEKE*, 2019, pp. 129–176.
- [23] J. Fischbach, A. Vogelsang, D. Spies, A. Wehrle, M. Junker, and D. Freudenstein, “Specmate: Automated creation of test cases from acceptance criteria,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 321–331.
- [24] Agile Alliance. (2021) Three Amigos. [Online]. Available: <https://www.agilealliance.org/glossary/three-amigos/>
- [25] —. (2021) INVEST. [Online]. Available: <https://www.agilealliance.org/glossary/invest/>
- [26] N. Santos, J. M. Fernandes, M. S. Carvalho, P. V. Silva, F. A. Fernandes, M. P. Rebelo, D. Barbosa, P. Maia, M. Couto, and R. J. Machado, “Using scrum together with uml models: A collaborative university-industry r&d software project,” in *Computational Science and Its Applications – ICCSA 2016*, O. Gervasi, B. Murgante, S. Misra, A. M. A. Rocha, C. M. Torre, D. Taniar, B. O. Apduhan, E. Stankova, and S. Wang, Eds. Cham: Springer International Publishing, 2016, pp. 480–495.
- [27] S. Mosser, M. Blay-Fornarino, and L. Duchien, “A commutative model composition operator to support software adaptation,” in *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kongens Lyngby, Denmark, July 2-5, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. Vallecillo, J. Tolvanen, E. Kindler, H. Störrle, and D. S. Kolovos, Eds., vol. 7349. Springer, 2012, pp. 4–19. [Online]. Available: [https://doi.org/10.1007/978-3-642-31491-9\\_3](https://doi.org/10.1007/978-3-642-31491-9_3)
- [28] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, *The Fuzzing Book*. Saarbrücken: CISPAA & Saarland University, 2019. [Online]. Available: <https://publications.cispa.saarland/3120/>
- [29] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, “Pycg: Practical call graph generation in python,” in *ICSE ’21: 43rd International Conference on Software Engineering*. ACM, 2021.
- [30] M. Nayrolles and A. Hamou-Lhadj, “CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects,” in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 153–164. [Online]. Available: <https://doi.org/10.1145/3196398.3196438>