



Thiago Delgado Pinto

**Unifying Agile Requirements Specification
Quality Control and Implementation
Conformance Assurance**

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Arndt von Staa

Rio de Janeiro
September 2018



Thiago Delgado Pinto

**Unifying Agile Requirements Specification
Quality Control and Implementation
Conformance Assurance**

Thesis presented to the Programa de Pós-graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the undersigned Examination Committee

Prof. Arndt von Staa

Advisor

Departamento de Informática - PUC-Rio

Prof. Marcos Kalinowski

Departamento de Informática - PUC-Rio

Prof. Alessandro Fabricio Garcia

Departamento de Informática - PUC-Rio

Prof. Leonardo Gresta Paulino Murta

UFF

Prof. Auri Marcelo Rizzo Vincenzi

UFSCar

Prof. Márcio da Silveira Carvalho

Vice Dean of Graduate Studies

Centro Técnico Científico – PUC-Rio

Rio de Janeiro, September 6th, 2018.

All rights reserved.

Thiago Delgado Pinto

He is currently professor at CEFET/RJ, where he teaches subjects related to Software Engineering. Graduated in Informatics from UNESA in 2003; obtained his Specialist degree in Software Engineering from the Senac-Rio University in 2010; and his Master's degree in Computer Science (Software Engineering) from the PUC-Rio in 2013.

Bibliographic data

Pinto, Thiago Delgado

Unifying Agile Requirements Specification Quality Control and Implementation Conformance Assurance / Thiago Delgado Pinto; Advisor: Arndt von Staa. – 2018.

252 f.: il. ; 30 cm

Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2018.

Inclui bibliografia.

1. Informática – Teses. 2. Agile. 3. Requirements specification. 4. Verification. 5. Validation. 6. Testing. 7. Generation. 8. Model-driven. I. Staa, Arndt von. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To Idaiane, for her encouragement and love at all times.

Acknowledgements

To my wife Idaiane, for the love, understanding, patience, encouragement, and support that made this work possible.

To my advisor, Arndt von Staa, for the counseling, pondering, fruitful discussions, reviews, and pleasant conversations.

To my friend Edgar Alexander, for the friendship during our travels between Nova Friburgo and Rio and all the moments we had to stay in Rio.

To my coworkers at CEFET/RJ, for backing me up during the last months of my research.

To all the companies involved with the multi-case study.

To all the members of the examining committee.

To all the professors at PUC-Rio who collaborate to my academic education or growth, especially to Arndt, Alessandro, and Simone.

To all the collaborators from the Informatics Department's administration office at PUC-Rio, in particular to Regina and Alex.

To PUC-Rio, CAPES, and CEFET/RJ, which partly supported this work, and for what I am grateful.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

Abstract

Pinto, Thiago Delgado; Staa, Arndt von (Advisor). **Unifying Agile Requirements Specification Quality Control and Implementation Conformance Assurance**. Rio de Janeiro, 2018. 252p. D.Sc. Thesis - Departamento de Informática, Pontifical Catholic University of Rio de Janeiro.

Agile requirements engineering practices are being used more commonly by software development teams. However, practices related to quality control still depend heavily on testers' expertise and manual labor, whilst produced requirements specifications are often imprecise and hard to verify statically by both stakeholders and computers. This thesis jointly tackles the problem of verifying statically agile requirements specifications and generating full-featured test cases and automated test scripts from them. Its main contributions include: (1) a new metalanguage, called Concordia, for writing agile requirement specifications that can be used for both verification and validation (V&V) activities involving stakeholders; (2) a novel approach to generate full-featured ready to use test cases and automated test scripts from the requirements specified with the metalanguage; (3) the assessment in industrial context of the approaches' ability to reduce risk of remaining defects and the costs of V&V.

Keywords

agile; requirements specification; verification; validation; testing; generation; model-driven;

Resumo

Pinto, Thiago Delgado; Staa, Arndt von (Orientador). **Unificando Controle de Qualidade de Especificação Ágil de Requisitos e Garantia de Conformidade de Implementação**. Rio de Janeiro, 2018. 252p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Práticas de engenharia de requisitos ágeis estão se tornando mais comuns em equipes de desenvolvimento de software. Contudo, as práticas relacionadas ao controle de qualidade ainda dependem fortemente do conhecimento, da experiência e do trabalho manual de testadores, em adição as especificações de requisitos produzidas são frequentemente imprecisas e difíceis de verificar estaticamente por interessados ou por algum computador. Essa tese ataca conjuntamente o problema de verificar estaticamente especificações de requisitos ágeis e de gerar casos de teste e scripts de teste automatizados completos a partir delas. Suas contribuições principais incluem: (1) uma nova metalinguagem, chamada Concordia, que permite escrever especificações de requisitos ágeis que podem ser usadas para atividades de verificação e validação (V&V); (2) uma nova abordagem para gerar casos de teste e scripts de teste automatizado completos, a partir de requisitos especificados com a metalinguagem; (3) a medição, em contexto industrial, da capacidade da abordagem em reduzir o risco de defeitos e custos de V&V.

Palavras-Chave

ágil; especificação de requisitos; verificação; validação; teste; geração; dirigida por modelos;

Table of Contents

1 Introduction	17
1.1. Motivation	17
1.2. Problem definition	19
1.3. Main contributions	24
1.4. Scope and constraints	25
1.5. Overview of the solution	25
1.6. Organization	26
2 Terms and Definitions	28
3 Validation and Verification from Agile DSLs	34
3.1. Related work	36
3.2. Research gaps	43
3.3. Concluding remarks	45
4 Agile DSLs and Metalanguages	46
4.1. Common DSLs	47
4.2. Specification of non-functional requirements	51
4.3. Integration with source code	52
4.4. Metalanguages	53
4.5. Concluding remarks	69
5 Restricted Natural Language Processing	70
5.1. Related work	70
5.2. Techniques	73
5.3. Approaches	78
5.4. Solutions for Natural Language Processing	80
5.5. Intent recognition with Bravey	83
5.6. Intent recognition in Concordia	86
5.7. Concluding remarks	88

6 Concordia	89
6.1. Language constructions	90
6.2. A quick example	104
6.3. Concluding remarks	105
7 Approach	106
7.1. Overview	106
7.2. High-level architecture	108
7.3. Verification	110
7.4. Validation	129
7.5. Maintenance	135
7.6. Concluding remarks	137
8 Proof of Concept	138
8.1. Selected cases	138
8.2. Detecting problems in specifications	140
8.3. Generating test cases and test scripts	145
8.4. Concluding remarks	170
9 Multi-case Study	171
9.1. Study design	171
9.2. Quantitative data	186
9.3. Qualitative data	194
9.4. Discussion	199
9.5. Concluding remarks	205
10 Epilogue	206
10.1. Conclusions	206
10.2. Future work	210
Bibliography	215
Appendix A – Architecture of the Solution	227
Appendix B – Concordia Grammar	244

List of Figures

Figure 1 - Main problem and subproblems	22
Figure 2 - Approach.....	25
Figure 3 – An Activity Diagram produced by Rane's approach.....	41
Figure 4 – Example of test cases produced by Rane's work.....	41
Figure 5 - Example of a Class Diagram generated by Soeken et al.'s work.....	72
Figure 6 – Example of an application of the Stanford Parser.....	72
Figure 7 – Example of a dependency analysis	75
Figure 8 - Example of a semantic grammar parse	79
Figure 9 - Overview of the process	106
Figure 10 – High-level architecture.....	109
Figure 11 - Prioritization techniques	113
Figure 12 - Number of inputs needed to detect defects.....	115
Figure 13 – Example of a T-wise combination	116
Figure 14 - Example of test scenarios	118
Figure 15 - Test case generation process.....	127
Figure 34 - Verification Case 1	140
Figure 35 - Verification Case 2	141
Figure 36 - Verification Case 3	142
Figure 37 - Verification Case 4	143
Figure 38 - Verification Case 5	144
Figure 39 – Some validations in the Login screen	145
Figure 40 - Execution of a test script.....	158
Figure 41 – Documentation artifacts and automated tests.....	187
Figure 42 - Usage of requirement specifications.....	187
Figure 43 - Features, maturity, and validation.....	188
Figure 44 –Concordia's reading comprehension.....	190
Figure 45 – Concordia's writing comprehension.....	191
Figure 46 - Perceived time writing Concordia specifications	191
Figure 47 - Perception about Concordia tests.....	192
Figure 48 - Overall perceptions on Concordia	194

List of Tables

Table 1 - Comparison criteria with related work	35
Table 2 - Comparison with related work.....	37
Table 3 - Comparison of Metalanguages	59
Table 4 - Tags in Penn Treebank.....	73
Table 5 - Selected relations from the Universal Dependency set.....	76
Table 6 – Solutions for Natural Language Processing	82
Table 7 - Language constructions in Gherkin and Concordia.....	90
Table 8 – Symbols in Concordia	92
Table 9 - Reserved tags	94
Table 10 - Database properties	97
Table 11 - UI Element properties	98
Table 12 - Test Events.....	103
Table 13 - Data test cases	120
Table 14 - Data test cases added according to declared properties	123
Table 15 - Compatibility between properties and data types	124
Table 16 - UI Element property compatibility	124
Table 17 - Strategies to mix data test cases.....	125
Table 18 - Roles and competences in requirements validation	131
Table 19 – Pre-test defect removal efficiency.....	132
Table 21 – Cases selected to exemplify problems detection.....	139
Table 22 - Cases selected to exemplify the produced tests	139
Table 23 - Companies' Profiles	172
Table 24 - Participants' Profiles.....	174
Table 25 - Questionnaire rationale	175
Table 26 – Semi-structured interviews rationale.....	182
Table 27 - Defects found per participant and company	193
Table 28 - Defects found in the prototype tool with the help of participants.....	199

Listings

Listing 1 – A test scenario from Rane’s work	40
Listing 2 – DSL for User Story	47
Listing 3 – Example of a Feature	47
Listing 4 – DSL for a Scenario	48
Listing 5 - Example of a Scenario	48
Listing 6 –Scenarios that vary by values	50
Listing 7 - Example of a Parameterized Scenario	51
Listing 8 - Example of NFR as a feature	52
Listing 9 – Step definition in Ruby	52
Listing 10 – Step definition in Java	52
Listing 11 – Example in JBehave	55
Listing 12 – Example in Gherkin	56
Listing 13 – Example in Robot.....	57
Listing 14 – Example in Gauge	58
Listing 15 – Example in Concordia.....	68
Listing 16 – Example of a semantic grammar	78
Listing 17 – Example in Bravery	84
Listing 18 – Example of parameterization in Bravery.....	85
Listing 19 – Example of a syntax rule for a intent	88
Listing 20 – Comment in Concordia	93
Listing 21 – Language in Concordia	93
Listing 22 – Import in Concordia	93
Listing 23 – Tags in Concordia	93
Listing 24 – Feature in Concordia	94
Listing 25 – Scenario in Concordia	95
Listing 26 – Constants in Concordia	95
Listing 27 – Table in Concordia.....	96
Listing 28 – Database in Concordia	96
Listing 29 – A simple UI Element in Concordia.....	99
Listing 30 – UI Element with Otherwise steps.....	99
Listing 31 – UI Element with dynamic properties	100

Listing 32 – Variant in Concordia	102
Listing 33 – Test Case in Concordia	103
Listing 34 – Test Events in Concordia	104
Listing 35 – A Quick Example in Concordia	104
Listing 36 – Feature Login	147
Listing 37 - Partial Test Cases produced for Login.....	152
Listing 38 - Partial Test Scripts for Login.....	157

List of Abbreviations

ATDD	Acceptance Test-Driven Development
ARE	Agile Requirements Engineering
ASD	Agile Software Development
BA	Business Analyst
BDD	Behavior-Driven Development
BFL	Business-Friendly Language
BNF	Backus-Naur Form
BRDSL	Business-Readable Domain Specific Language
CSS	Cascading Style Sheets
DSL	Domain-Specific Language
GUI	Graphical User Interface
HTML	Hyper-Text Markup Language
JSON	JavaScript Object Notation
MBT	Model-Based Testing
NFR	Non-Functional Requirement
RE	Requirements Engineering
SbE	Specification by Example
SM	State Machine
SQL	Structured Query Language
SUT	System Under Test
TDD	Test-Driven Development
UAT	User Acceptance Testing
UI	User Interface
UL	Ubiquitous Language
UML	Unified Modeling Language
V&V	Verification and Validation
XML	eXtensible Markup Language
XP	eXtreme Programming

The hardest part of the software task is arriving at a complete and consistent specification, and much of the essence of building a program is, in fact, the debugging of the specification.

Freddy Brooks, in “No silver bullet” (1986)

1Introduction

It is easier to change the specification to fit the program than vice versa.

- Alan Perils (Turing award-winning, 1966)

Despite the need for software is increasing every day, small and medium software companies still struggle to benefit from state-of-the-art techniques of software development and testing. The pressure for deadlines, the difficulty of applying theory, and the low technical training of software teams make such techniques intangible or hard to adopt. The advent of agile practices and frameworks, such as Lean Software Development (POPPENDIECK, 2007) and Behavior-Driven Development (BDD) (NORTH, 2006), compensate the lack of formal software engineering methods with frequent customer feedback, test automation, essential documentation, and focus on deliverables that bring value to the business. Nevertheless, many problems persist. The quality and coverage of produced tests depend heavily on testers' expertise and manual labor; produced requirements specifications are often imprecise and mix business and computing jargons, making them both difficult to be statically validated by customers and other stakeholders (aiming to reduce errors, imprecisions, inconsistencies, or incompleteness), and vague enough to not be useful for testers and developers. These problems increase the chances of delivering wrong or buggy software and the risk of useless rework. *How can we mitigate them?*

1.1.Motivation

It is well established that removing a defect corresponding to a software requirement at the beginning of its construction can be up to ten times cheaper than do it before construction and up to a hundred times cheaper than do it after launching (BOEHM & TURNER, 2003a; BOOCH, 1999; FAGAN, 1976; JONES, 1996; LEFFINGWELL, 1997; SHULL et al., 2002). Debugging and fixing software are costly activities, which may correspond to up to fifty percent of the time of a software project (BOEHM & BASILI, 2001; JONES, 1998; MILL & WEINBERG,

1988; SHULL et al., 2002; WHEELER; BRYKCYNSKI & MEESON JR, 1996; WIEGERS, 2002; WIEGERS & BEATTY, 2013). Pre-test defect removal practices, such as formal inspections and static analysis, may cut development and maintenance costs by about thirty percent (JONES & BONSIGNOUR, 2012; MCCONNELL, 2004). To invest in validation practices like these, as well as in “correctness by construction” (AMEY, 2002) and in software testing automation may contribute substantially to help projects to stay on budget and schedule (AMEY, 2002; JONES & BONSIGNOUR, 2012; MCCONNELL, 2004). Although there are clear benefits of adopting effective verification and validation (V&V) practices, and there are a plethora of approaches in the literature, there still exists a big gap between real software systems¹ and the practical usability of techniques proposed by the research community (ANAND et al., 2013).

In the past few decades, a considerable amount of research effort has been spent on V&V activities individually. Approaches to *both* deal with V&V are still meager and have important open challenges (ANAND et al., 2013; DUBOIS et al., 2013). Dubois *et al.* (2013), for instance, point out some challenges and related questions, of which we highlight:

- i) *Gap between models and V&V formalisms*: How do we express properties at the level of models in a way understandable to clients? How do we formulate models and properties in a single language transparent to clients? How do we report the V&V results and diagnostics in an appropriate form to clients? How do we bridge the gap between formally expressed and verified properties on one side and client attention on the other side?
- ii) *Informal vs. formal vs. incomplete modeling*: How do we handle incomplete or partial models in relation to V&V?
- iii) *Comparison and benchmarking*: How do we compare existing V&V tools employed for modeling with relation to functionality, coverage, scalability, expressiveness, executing system (*i.e.*, for models at runtime)? Which criteria are appropriate for comparison?

¹ Software systems produced in industry or academia to solve real-world problems.

- iv) *Domain-specific languages*: How can Domain-Specific Languages (DSL) be defined so that they are close to the domain concepts on the one hand, but still allow the generation of meaningful input files for verification tools? How do we express the properties to be verified at the domain level in a user-friendly way? Can specifications be integrated with the same DSL or model used for describing the to-be-verified system without creating self-fulfilling prophecies? How can we lift the result of a verification (*e.g.*, an example program execution that demonstrates the failure) back to the domain level and express it in terms of the DSL-level input? Can incremental language extensions help to make programs expressed in general-purpose languages more checkable?

Since such challenges hinder the practices above to effectively reducing costs and time in software projects, new approaches to mitigating problems related to the V&V are needed and welcome.

Nowadays, agile software development (ASD) is used to cope with the increasing complexity in system development (SCHÖN; THOMASCHEWSKI & ESCALONA, 2017). A growing number of software companies is adopting “Agile” requirements engineering (RE) practices as a way to solve problems of traditional RE – *e.g.*, communication issues, requirements validation, requirements documentation (CURCIO et al., 2018; INAYAT et al., 2015; SCHÖN; THOMASCHEWSKI & ESCALONA, 2017). *Are there integrated approaches for V&V that consider the agile software development? Do they try to mitigate the aforementioned challenges? How they try to reduce the risks of delivering wrong or buggy software and the risk of useless rework?*

1.2.Problem definition

Correct software is an important research goal in Software Engineering and is a permanent aim of any software company. Defects arising from incorrect or incomplete requirements specifications are admittedly expensive (BOEHM & TURNER, 2003b; BOOCH, 1999; FAGAN, 1976; JONES, 1996; LEFFINGWELL, 1997; SHULL et al., 2002) and may lead to software that does not fit stakeholders’ needs. On the other side, software that can meet their needs, may not work properly due to

defects arising from incorrect development and scarce or inadequate testing. Thus, verifying the specification, validating its requirements with stakeholders and checking (testing) properly whether the produced software complies with the requirements are imperative activities to create correct software. Furthermore, the availability of execution examples – *i.e.*, test scripts – previous to initiating development is expected to reduce the number of defects inserted into the software due to incorrect understanding of the specification (ADZIC, 2009, 2011; GÄRTNER, 2012). Providing ways to reducing adequacy and quality control costs is a demanding challenge to pursuit (ANAND et al., 2013; DUBOIS et al., 2013).

IEEE 26515 (2012) affirms that “*In agile development, it is important that the development of the user documentation is part of the same processes as the software product lifecycle, and performed in conjunction with the development of the software. This enables the software and the user documentation to be tested, distributed, and maintained together. In agile development, the software cannot be considered complete without the production and validation of the associated user documentation.*”. These practices are also fostered by methodologies like Acceptance Test-Driven Development (ATDD) (GÄRTNER, 2012), Behavior-Driven Development (BDD) (NORTH, 2006), and Specification by Example (SbE) (ADZIC, 2009). All of them use user documentation as a primary artifact to discuss and validate requirements with stakeholders, for creating a shared understanding between stakeholders and the software team, and to derive tests that help to verify the compliance of the produced software with these requirements. Systematic mapping studies on agile software development (CURCIO et al., 2018; INAYAT et al., 2015; SCHÖN; THOMASCHEWSKI & ESCALONA, 2017) identify that *user story* is the most common format to write such user documentation. This holds true in the aforementioned methodologies. User stories capture the needs and desires of the involved stakeholders in the form of *features* and *scenarios* (ADZIC, 2009, 2011; NORTH, 2003). Features and scenarios are *business readable* DSLs (FOWLER, 2008), *i.e.*, they are readable by business people and, thus, *proper for validation*. However, *are these “Agile DSLs” proper for verification?* Current tools to support ATDD, BDD, and SbE, such as Cucumber (HELLESØY, 2009) and JBehave (NORTH, 2003), only generate *test script skeletons* from these Agile DSLs, *i.e.*, the software team still have to produce their content manually. Current approaches for

producing test cases from Agile DSLs – *e.g.*, Rane (2017), Elghondakly *et al.* (2015), Kamalakar *et al.* (2013) – cannot produce test data and test oracles, and cannot produce test scripts that verify whether a software implementation correspond to its specifications (*e.g.*, functional tests). Furthermore, there are no approaches concerned with both V&V activities, *e.g.*, none of them is concerned with pre-test defect removal practices, such as static analysis, to identify problems in requirements specifications prior to testing. *Is it possible to provide an integrated approach for ASD that tries to mitigate these gaps?* A possible reason for the lack of such approaches is the difficulty to make a computer to understand requirements in natural language, due to the enormous variation on their writing style. Without restricting the writing style and the adopted vocabulary, the problem can turn into an undecidable problem. Approaches that tried somehow to restrict the input format were able to extract the needed data for generating their output. *Is it possible to do the same for Agile DSLs? How can Natural Language Processing (NLP) help with that?*

Other important *sub-problems* to consider since we want to generate *full-featured* functional test cases and test scripts – *i.e.*, test cases and test scripts with relevant data and oracles – are (ANAND *et al.*, 2013; BARR *et al.*, 2015; LIU *et al.*, 2014):

- 1) *Combinatory explosion* – the difficulty to verify all the paths in an application. Today’s applications are getting bigger and bigger, and solutions that do not consider ways to adequately balance time and coverage can be unfeasible in practice. *Which reduction criteria can be considered to make the test generation feasible in an integrated solution for V&V? Can these criteria use any information from the specification?*
- 2) *Choosing the relevant test data*: model-based automatic test data generation requires analyzing constraints and producing the right input values to satisfy them – or to not satisfy them –, as well as to predict the associated execution paths. When a constraint is unsatisfiable, the corresponding path is unfeasible. The analysis for choosing test data may involve symbolic execution.

How can we generate relevant test data from Agile DSLs? How can we avoid path explosion? How to detail with complex constraints?

- 3) *Generating test oracles*: test oracles must distinguish whether a certain system behavior is correct (or not). In manual testing, a human plays the oracle role. Computer-generated oracles should try to eliminate human intervention. *How to produce correct and relevant oracles from Agile specifications?*

1.2.1.Summary

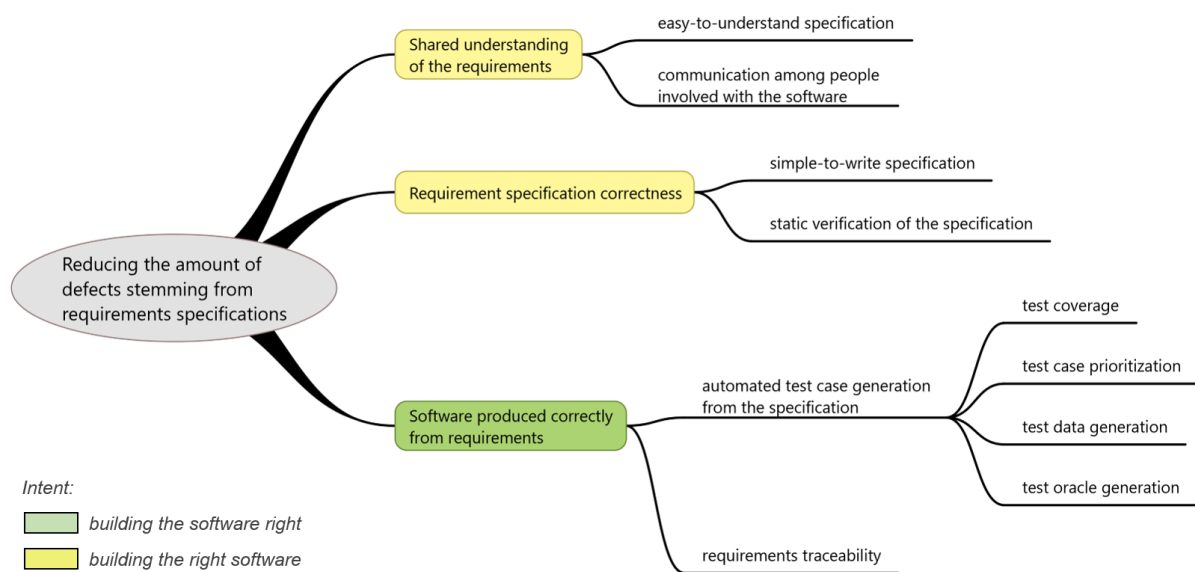


Figure 1 - Main problem and subproblems

Figure 1 tries to summarize the main problem and sub-problems tackled by this thesis – although there are more involved. As mentioned, the approach involves V&V techniques. Regarding *validation*, to find a way of providing a specification that is easy-to-understand by stakeholders which are not fluent in computational or application aspects, simple-to-write by the software team, and able to be verified statically. Regarding *verification*, to find a way of helping software teams to check whether their implementation of a system is in accordance with the requirements specification, through tests, and a way of checking requirements for (syntactic, semantic, logic) errors.

The approach should also try to mitigate the following *sub-problems*:

- (i) Avoiding combinatory explosion, aiming to find a balance between coverage and feasibility;
- (ii) Choosing the relevant test data;
- (iii) Generating test oracles;
- (iv) Providing traceability between generated test scripts and requirements specifications.

1.2.2.Proposal

We propose an approach to reduce the number of defects stemming from incorrect or incomplete requirements specifications – so costly and longstanding – and reducing the costs of producing acceptance tests. The approach introduces a business-readable, Agile-friendly, state-based, statically verifiable, requirements specification metalanguage as the base model to verification and validation (V&V) activities. It also provides means to use the metalanguage for generating traceable, full-featured test cases and test scripts that can mitigate the aforementioned approaches' limitations.

Therefore, this thesis investigates the use of a metalanguage based on Agile DSLs to tackle the problems summarized in Figure 1. *More specifically, whether requirements specified with this metalanguage can be used for validation with stakeholders and whether it can be used for preventing or detecting defects.*

1.2.3.Research questions

The main research question (MRQ) of this work is:

MRQ: *Can Agile DSLs combined with our approach serve for both validating and automatically verifying applications effectively?*

Secondary research questions (SRQ) arisen from the MRQ are:

SRQ1: *How can Agile DSLs be used for generating full-featured test scripts?*

SRQ2: *Can test scripts generated from Agile DSLs reveal defects in existing applications?*

SRQ3: *Can an approach for V&V that uses Agile DSLs reduce test time and costs?*

SRQ4: *Can an approach for V&V that uses Agile DSLs be used for preventing defects?*

1.2.4.Evaluation

We evaluated the proposed approach through a multi-case study with small software companies. They received an initial one-day training about the metalanguage and the prototype tool, and were supported during the case studies to make sure they could use the approach with their applications. We collected quantitative data (*e.g.*, number of features involved, maturity level of applications and features) and qualitative data (*e.g.*, participants' opinions and observations about the language and generated tests) through questionnaires and semi-structured interviews. Results are detailed in chapter 9.

In chapter 8, we present a proof of concept to illustrate the proposed approach's capacity to check errors in requirements specifications written with Concordia – the introduced metalanguage – and its capacity to detect differences between an application and a system under test.

1.3.Main contributions

Main contributions are:

- (i) a new metalanguage for writing agile requirement specifications that can be used for both V&V activities;
- (ii) the first approach to generate *full-featured* ready to use test cases and test scripts from agile requirements specifications;
- (iii) the first *integrated* approach for V&V of agile requirements specifications;
- (iv) the assessment in industrial context of the proposed approach.

Chapter 10 presents a detailed list of the contributions.

1.4. Scope and constraints

We are currently narrowing our approach to *information systems*, due to research time restrictions to verify other contexts.

The approach can only generate *functional* test cases. Although the presented metalanguage, Concordia (chapter 6), can be used for specifying functional and non-functional requirements, our approach cannot generate test cases for non-functional requirements (NFR). The automatic test case generation from NFR consists of a challenging problem to be resolved (ANAND et al., 2013), even for those NFR verifiable via software, such as performance, security, availability or portability. Despite that, since Concordia is compatible with the Gherkin metalanguage (section 4.4.2), it can be used by Gherkin-based tools like Cucumber or JBehave to generate *test script skeletons* from NFR.

1.5. Overview of the solution

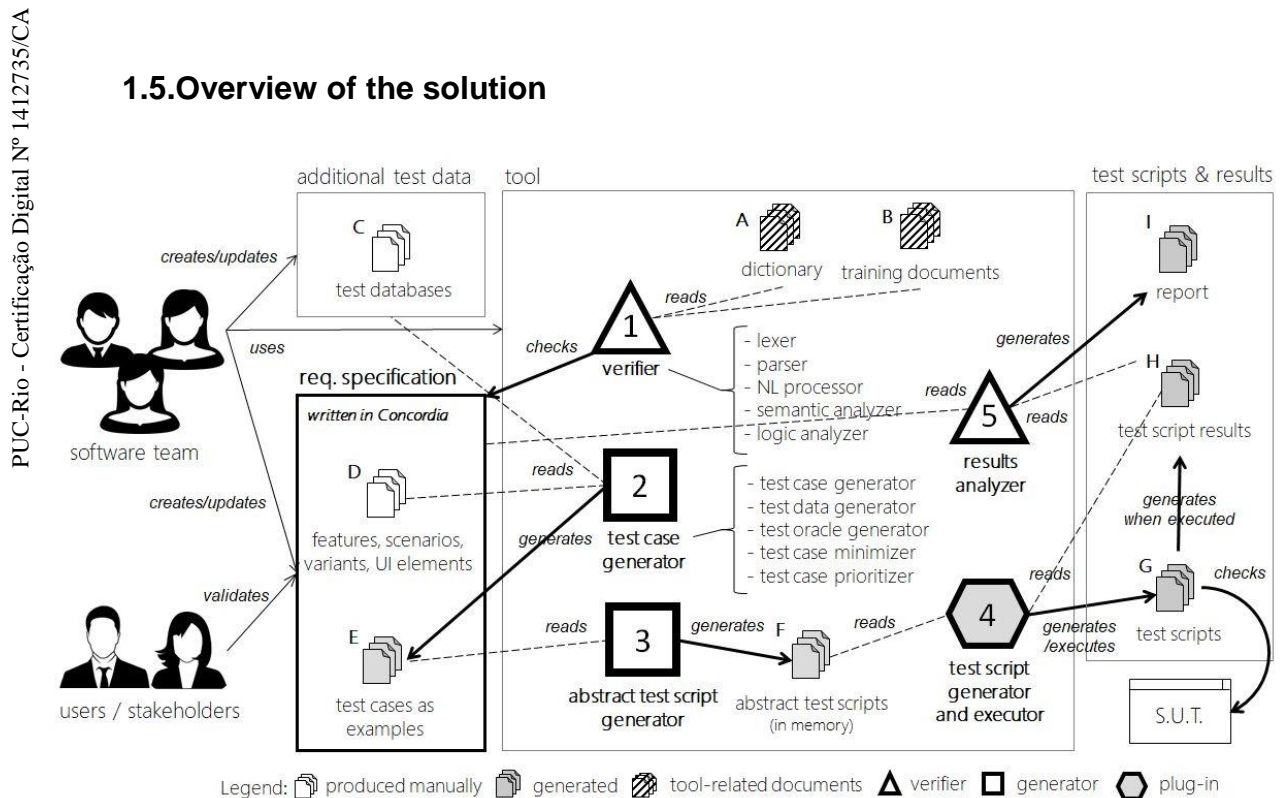


Figure 2 - Approach

Figure 2 illustrates the proposed approach. The software team produces the requirements specification collaboratively using Concordia, the introduced meta-language. For example, Business Analysts (BA) write Features and Scenarios based on business needs and desires; User Interface (UI) Designers, Testers, and Developers write *Variants* – *i.e.*, templates of expected interactions with the system’s user interface – and detail business rules related to the UI together with BA. The team uses the tool to check the specification for errors or inconsistencies and to generate test cases. Users and Stakeholders validate Features, Scenarios, and Variants – and Test Cases when needed –, and the software team makes adjustments in the specification based on their feedback. The software team then uses the tool to generate test scripts from the requirements specification and run them for checking the compliance of implemented features with the specified functional requirements and for discovering defects. Fixes in the application or adjustments in the application or in the specification are eventually made. This entails a need to revise the specification and to generate a new version of the test scripts. Finally, when the application is released, the team validates it with users and stakeholders. New adjustments or features are reflected in the specification and the process restarts.

Concordia is composed of Agile DSLs (chapter 4 details them). Agile DSLs usually adopt some words as prefixes and let the rest of the sentences be written in natural language. Since their structure is not formal enough for generating test cases automatically, the approach uses natural language processing (NLP) techniques (chapter 5) and adopts a restricted (although flexible and adaptable) vocabulary.

1.6. Organization

The rest of this work is organized into the following chapters:

2. *Terms and Definitions*: presents terms and definitions used in the rest of this thesis;
3. *Validation and Verification from Agile DSLs*: describes and compares works related to this thesis and related research gaps;
4. *Agile DSLs and Metalanguages*: details commonly used Agile DSLs and metalanguages, and compares these metalanguages with that introduced by this thesis;

5. *Restricted Natural Language Processing*: presents techniques for NLP, compares industry-level solutions for NLP, and details the techniques adopted in the proposed approach;
6. *Concordia*: presents the novel metalanguage to specify software requirements based on Agile DSLs;
7. *Approach*: details the approach proposed in this thesis;
8. *Proof of Concept*: illustrates the approach's capacity to check problems in Concordia specifications, and differences between these specifications and a system under test.
9. *Multi-case Study*: details the multi-case study with software companies, and discusses its results.
10. *Epilogue*: presents the contributions and proposes some future work.

Additional content includes:

- *Appendix A – Architecture of the Solution*: Details the architecture of the proposed solution.
- *Appendix B – Concordia Grammar*: Grammar of the Concordia metalanguage in Backus-Naur Form (BNF).
- *Appendix C – Static Checking*: Presents a listing with the static verifications performed by the proposed approach.

2Terms and Definitions

I welcome new words, or old words used in new ways, provided the result is more precision, added color or greater expressiveness.

- William Safire (American writer)

This chapter presents the terminology adopted in the thesis.

A *domain-specific language* (DSL) is a computer programming language of limited expressiveness focused on a particular domain (FOWLER, 2009). This work uses the term DSL as a synonym for a textual, *grammar-based DSL*. Model-driven engineering (MDE) community also uses the term DSL as a synonym for *domain-specific modeling language* (DSML), which may use meta-models to describe syntax (instead of a grammar) and present different semantics (KOSAR; BOHRA & MERNIK, 2016). Examples of (grammar-based) DSLs include HTML, CSS, SQL, YACC grammars for creating parsers, GraphViz's Dot (graphical rendering of node-and-arc graphs), R (a language and platform for statistics), Matlab (for numerical and symbolic computing), JMock (library for defining mock objects in test scripts), Unix shell scripts, regular expressions, etc. According to Mernik *et al.* (2005), DSLs are also called *application-oriented* (SAMMET, 1969), *special purpose* (WEXELBLAT, 1981), *specialized* (BERGIN; GIBSON & PRESS, 1996), *task-specific* (NARDI, 1993), or *application* (MARTIN, 1985) languages. A systematic mapping study on DSLs is presented by Kosar *et al.* (2016).

A *business-readable DSL* (BRDSL) is one which business people can read and understand (FOWLER, 2008). Such languages are not necessarily *business-writable* – that is, they are not intended to be directly used by business people, but rather by business analysts or software developers, who translate the business knowledge to the target language. Usually, a BRDSL facilitates a software team to use a ubiquitous language.

A *ubiquitous language* (UL) is a language structured around the domain model, that can be used by all team members in a bounded context to connect all the activities of the team with the software (EVANS, 2003). It can be used in conversations with domain experts to foster domain understanding and to avoid terms that are awkward or inadequate to their communication, such as computer jargon. A clear communication prevents misunderstandings about requirements and, thus, may reduce the number of defects originated from them.

A *defect* (a.k.a. *fault*) is a fragment of an artifact that, when used or executed, may lead to an error (ISO; IEC & IEEE, 2017; STAA, 2017). An *error* is a deviation between what is desired or intended and what is specified, required or expected (ISO; IEC & IEEE, 2017; STAA, 2017). For instance, an error could be omission or misinterpretation of user requirements in a software specification, incorrect translation, or omission of a requirement in the design specification (ISO; IEC & IEEE, 2017). Errors are caused by defects. A *failure* is an observed error (STAA, 2017). Thus, the occurrence of an error is always unknown until the error is observed by some means. *Error latency* is the elapsed time until an error becomes a failure (STAA, 2017). *Failure Detection Rate* (FDR) (a.k.a. *Fault Detection Rate*) is the number of failures per unit of time, detected by a test suite (ELBAUM; MALISHEVSKY & ROTHERMEL, 2002). Defects can appear in any artifact. For example, a requirements specification may contain defects. Exercising them, *i.e.*, developing in conformance to them will produce erroneous designs or architectures. If these errors are not identified, they may propagate into other artifacts, ultimately into code and test suites. Many specification defects correspond to inadequacies, *i.e.*, to specifications that do not conform to stakeholders needs. Furthermore, such defects may not be observed even when formal specifications are used.

A *test oracle* – from now on referred just as *oracle* – is a person or mechanism that can determine whether the output produced by a given input to some artifact is correct or not (WEYUKER, 1982). Computer-generated oracles try to mimic the human ability to observe failures. However, a human tester cannot always predict all the expected outputs for an input with complex data. Sometimes *partial oracles* may be produced for providing a verdict for these outputs (*e.g.*, using inference models) (FINOT et al., 2013). Relations among properties of inputs and outputs of

multiple executions, called *metamorphic relations*, may also serve to detect defects (LIU et al., 2014).

Verification and *validation* (V&V) are confirmations, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled (ISO; IEC & IEEE, 2017). In the context of software systems, *verification* is a way to decide whether a team is building the product right, while *validation* is a way to decide whether a team is building the right product (BOEHM, 1984; ISO; IEC & IEEE, 2017). Usually the former is performed by the team and the latter by users (customers), with or without the team (SOMMERVILLE, 2011). Verification may occur statically or dynamically. *Static verification* can be performed through (formal) *inspections* (BRIAND et al., 1998; LAITENBERGER, 2002; WHEELER; BRYKCZYNSKI & MEESON JR, 1996), *revisions* (CIOLKOWSKI et al., 2002; CIOLKOWSKI; LAITENBERGER & BIFFL, 2003), or *static analysis* of the model (*i.e.*, by using a software tool) (BLASCHEK, 1985; LANDI, 1992; ZHENG et al., 2006). *Dynamic verification* is usually performed through tests.

Data-driven testing is a technique that consists of storing test data separately from the sequence of actions (ISO; IEC & IEEE, 2016). For one test procedure with a defined sequence of actions, multiple sets of data can be provided. The sequence of actions is then executed for each of the sets of data. Depending on the implementation, the data is either stored in a table, spreadsheet or database. Data-driven testing is an option to decouple the parameters from the test.

Keyword-driven testing is a way of describing test cases by using a predefined set of keywords (ISO; IEC & IEEE, 2016). These keywords are names that are associated with a set of actions that are required to perform a specific step in a test case. The fundamental idea is using these keywords to create manual or automated test cases without requiring detailed knowledge of programming or test tool expertise. The vocabulary included in these dictionaries or libraries of keywords is, therefore, a reflection of the language and level of abstraction used to write the test cases, and not of any standard computer programming language. Related benefits include

ease of use, maintainability, test information reuse, and potential cost and schedule savings (ISO; IEC & IEEE, 2016).

Functional testing (a.k.a. *black-box testing*) is a testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions (ISO; IEC & IEEE, 2017). It can be conducted to evaluate the compliance of a system or component with specified functional requirements (ISO; IEC & IEEE, 2017).

Regression testing is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements (ISO; IEC & IEEE, 2017).

Acceptance testing is conducted to determine whether a system satisfies its *acceptance criteria* (set of stakeholder required conditions) and to enable the customer or stakeholder to determine whether to accept or not the system (ISO; IEC & IEEE, 2017).

Acceptance Test-Driven Development (ATDD) is a practice in which the whole team collaboratively discusses acceptance criteria, with examples, and then distills them into a set of concrete acceptance tests before development begins (ADZIC, 2009; HENDRICKSON, 2008a).

Behavior-Driven Development (BDD) is a practice that uses conversations around concrete examples of system behavior to help understanding how features will provide value to the business. BDD encourages business analysts, software developers, and testers – usually called “*the three amigos*” – to collaborate more closely by enabling them to express requirements in a more testable way, in a form that both the development team and business stakeholders can easily understand (SMART, 2014).

Specification by Example (SbE) is described by Gojko Adzic (2009) as a superset of practices that include ATDD and BDD. He points out that ATDD focuses on clearing the targets for development, creating automated tests, and preventing

functional regression, while BDD focuses on the process of specifying scenarios of system behavior and building a shared understanding between stakeholders and delivery teams through collaboration and clarification of specifications – although BDD also considers functional regression an important thing. The key process patterns adopted by SbE are discussed in Adzic’s book (2011).

A *stakeholder* is a role, an individual or an organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations (ISO; IEC & IEEE, 2017). In Scrum, it is the *Product Owner* (SCHWABER, 2004). In Extreme Programming, it is the *Customer* (BECK, 2003). A stakeholder usually defines the features to be implemented in terms of externally verifiable behavior whilst the implementation team decides on the internal implementation details (HENDRICKSON, 2008b). In most cases when stakeholders are mentioned, they are referring to *business* stakeholders, *i.e.*, those interested or involved with the business for which the system is being developed or is related with. Despite, stakeholders may include supporters, trainers, maintainers, supplier organizations, regulatory bodies, and any other interested parties (ISO; IEC & IEEE, 2017). Considering the multitude of physical stakeholders, the role or roles performed by them are what matters, instead of their individual desires. Some stakeholders can have interests that conflict with others. Business Analysts, System Analysts, and System Architects may work together to reconcile these opposed interests.

A *feature* is a functional or non-functional distinguishing characteristic of a system that end-users and other stakeholders can understand (ISO; IEC & IEEE, 2015, 2017). In BDD terms, a feature is a fragment of software functionality that helps users or other stakeholders to achieve some business goal (SMART, 2014). These definitions are complementary and not conflicting, and this work will adopt the sum of their meanings.

A *user story* is a simple narrative illustrating the user goals that a software function will satisfy (ISO; IEC & IEEE, 2017). This narrative may be a description of a software requirement, function, feature, or quality attribute.

A *scenario* is a step-by-step description of a series of events that occur concurrently or sequentially. It can be a user story, use case, operational concept, or sequence of events the software may encounter (ISO; IEC & IEEE, 2017). In BDD, a feature may contain one or many scenarios that describe contexts in which it is used. These contexts are usually subjected to validation by stakeholders. A *successful scenario* (a.k.a. “happy scenario”) is one that produces its postconditions, due to all inputs are considered correct. An *error handling scenario* is one that does not produce its postconditions due to one or more inputs – actions or data – considered invalid are used and the system under test (SUT) criticizes them rather than completing the task. A *failing scenario* is one that does not produce its postconditions due to unexpected defects.

A *business rule* is an independent formalism to represent business logic, (HALLE, 2002) *i.e.*, its rule defines or constraints some aspect of the business. A *system rule* defines or constraints some aspect of the intended system. A system rule may represent or implement a business rule. In this work, *we may use them as synonyms*.

Agile development is a software development approach based on iterative development, frequent inspection and adaptation, and incremental deliveries, in which requirements and solutions evolve through collaboration in cross-functional teams and through continuous stakeholder feedback (ISO; IEC & IEEE, 2012).

We define as *Agile DSLs* those business-readable DSLs used by Agile companies (*i.e.*, companies that adhere to Agile methodologies) to specifying requirements, to discussing them with stakeholders, to developing features, or to producing test cases or test scripts.

3Validation and Verification from Agile DSLs

Incorrect documentation is often worse than no documentation.

- Bertrand Meyer

This chapter presents and compares approaches directly related to this thesis.

We analyzed approaches that use **natural language specifications (NLS) written with Agile DSLs** for producing test scenarios, test cases or test scripts, or for performing some type of static validation of such requirements. Much research has been undertaken for extracting conceptual models from NLS written with Agile DSLs (chapter 4 details these DSLs). These conceptual models include OWL ontologies, UML diagrams, Data Flow Diagrams (DFD), and Entity-Relationship Diagrams – *e.g.*, (ROBEER et al., 2016; SOEKEN; WILLE & DRECHSLER, 2012; VIDYA SAGAR & ABIRAMI, 2014; YUE; BRIAND & LABICHE, 2011). Just a few works, however, propose approaches that deal with V&V activities – the next section details them.

We defined a set of criteria to facilitate the comparison of approaches with ours'. Table 1 presents these criteria and section 3.2 discuss some gaps that they may represent. Criterion 18 concerns with the generation of *relevant test cases*, while criterion 23 concerns with the generation of *relevant test oracles*. We define *relevant test cases* as those that use testing techniques recognized as effective by literature, such as Equivalence Class Partitioning, Boundary Value Analysis, and Random Values (MYERS; THOMAS & SANDLER, 2011). We detail this subject in section 7.3.4. We define *relevant test oracles* as those that can check the expected behavior when input data are considered invalid, according to the specification (we detail this subject in the section 7.3.6).

Table 1 - Comparison criteria with related work

#		Criterion	Reason
1	Spec.	Input	Evaluate the artifacts used as input
2		Agile DSLs	Indicate the adopted Agile DSLs
3		Input from plain-text files	Plain-text files do not require special tools and are easy to use concurrently by software teams with Version Control Systems (<i>e.g.</i> , Git, Subversion).
4		Support for more than one spoken language	Evaluate if the specification is tied to a spoken language
5		Used to minimize the generation of test cases	Evaluate if there is test case minimization, which is important to make the execution time feasible
6		Validation with stakeholders is analyzed	Evaluate if the approach analyzes the validation of the specification with stakeholders
7		Static validation is addressed	Evaluate if the approach analyzes the static validation of the specification
8	Test Cases	Generate test cases	Evaluate if the approach generates test cases.
9		Accept test cases as input.	Evaluate if the approach accepts test cases as input.
10		Input from plain-text files	Plain-text files do not require special tools and are easy to use concurrently by software teams with Version Control Systems (<i>e.g.</i> , Git, Subversion).
11		Support for more than one spoken language	Evaluate if the specification is tied to a spoken language
12		Can combine test scenarios of the same feature	Evaluate if the approach can combine test scenarios from the same feature
13		Can combine test scenarios of different features	Evaluate if the approach can combine test scenarios from different features
14		Covered test scenarios	Indicate the covered test scenarios
15		Coverage criteria	Evaluate if there are coverage criteria
16		Accept test data	Evaluate if the test cases accept test data
17		Generate test data	Evaluate if the approach generates test data
18		Generated test data cover relevant cases	By covering important cases, test cases increase the chances to discover defects
19		Can use test data from external data sources	Evaluate if the test data can be loaded from external data sources
20		Can minimize test data from external data sources	Evaluate if external test data can be filtered somehow

#		Criterion	Reason
21		Can define restrictions based on external data sources	Evaluate if external test data can be used to define business rules or restrictions that will be used to generate test data
22		Generate test oracles	Evaluate if the approach generates test data
23		Generated test oracles cover relevant cases	By covering important cases, test oracles increase the chances to discover defects
24		Used to prioritize the generation of test scripts	Evaluate if test cases are used to prioritize the generation of test scripts. This can reduce execution time.
25	Test Scripts	Generate test scripts	Evaluate if the approach generates test scripts automatically
26		Verify the compliance with requirements	Evaluate if the test scripts verify the compliance of the SUT with the specified requirements
27		Have test data	Evaluate if the test scripts have test data.
28		Have test oracles	Evaluate if the test scripts have test oracles
29	TS Exec.	Executes automatically	Evaluate if the approach can execute test scripts automatically.
30		Minimizes the number of executed test scripts	Evaluate if the approach can reduce the number of executed test scripts and, thus, decrease execution time.
31		Analyze test script execution results	Evaluate if the approach can analyze execution results and track their relation with the specification

3.1.Related work

Table 2 compares related work according to the criteria detailed in Table 1. Letters were used to shorten their identification, due to space restrictions in the table: **R** for **Rane (2017)**; **E** for **Elghondakly et al. (2015)**; and **K** for **Kamalakar et al. (2013)**.

Table 2 - Comparison with related work

#		Criterion	R	E	K	This work
1	Spec.	Input	Features, test scenarios, user-defined dictionary	Features, scenarios	Features, scenarios	Specifications in Concordia (see chapter 6)
2		Agile DSLs	User story	User story, scenario	User story, scenario	User story, scenario, and others (see 4.4.5)
3		Input from plain-text files	No	Yes	Yes	Yes
4		Support for more than one spoken language	No	No	No	Yes
5		Used to minimize the generation of test cases	No	No	No	Yes
6		Validation with stakeholders is analyzed	No	No	No	Yes
7		Static validation is addressed	Poorly (if user stories follow the DSL “As I/I want/So that”)	Maybe, it mentions using symbolic evaluation but it does not offer details	No, it just evaluates generated sentences	Yes, see 7.4.2

#		Criterion	R	E	K	This work
8	Test Cases	Generate test cases	No	Yes	No	Yes
9		Accept test cases as input.	Yes	No	No	Yes
10		Input from plain-text files	No	N/A	N/A	Yes
11		Support for more than one spoken language	No	No	N/A	Yes
12		Can combine test scenarios of the same feature	N/A	Unknown	N/A	Yes
13		Can combine test scenarios of different features	N/A	Unknown	N/A	Yes
14		Covered test scenarios	All test case paths	Unknown	N/A	All Variant (6.1.11) paths, state-based paths, all constraints of UI Elements (6.1.10), plus a set data test cases (see 7.3.5)
15		Coverage criteria	Paths of test scenarios	Unknown	N/A	Paths of Variants, state-based references, constraints of UI Elements (see 7.3.4)
16		Accept test data	No	No	N/A	Yes
17		Generate test data	No	No	N/A	Yes
18		Generated test data cover relevant cases	N/A	N/A	N/A	Yes
19		Can use test data from external data sources	No	N/A	N/A	Yes
20		Can minimize test data from external data sources	N/A	N/A	N/A	Yes
21		Can define restrictions based on external data sources	N/A	N/A	N/A	Yes
22		Generate test oracles	No	N/A	N/A	Yes

#		Criterion	R	E	K	This work
23		Generated test oracles cover relevant cases	N/A	N/A	N/A	Yes
24		Used to prioritize the generation of test scripts	No	N/A	N/A	Yes
25	Test Scripts	Generate test scripts	No	No	Yes	Yes
26		Verify the compliance with requirements	N/A	No	No	Yes
27		Have test data	N/A	N/A	Yes	Yes
28		Have test oracles	N/A	N/A	Yes	Yes
29	TS Exec.	Executes automatically	N/A	N/A	Unknown	Yes
30		Minimizes the number of executed test scripts	N/A	N/A	Unknown	Yes
31		Analyze test script execution results	N/A	N/A	No	Yes

Rane (2017) presents an approach for producing test cases in English language and Activity Diagrams from *features* written in Gherkin (section 4.4.2), test scenario descriptions, and a dictionary of synonyms. All the input is given through a graphical user interface. Test scenario descriptions adopt a format based on period-terminated sentences, like in the example from Listing 1.

Listing 1 – A test scenario from Rane’s work²

```
User inserts Card.
User enters PIN.
IF user authorized THEN Select Account Type ELSE Eject card.
Y: Enter Amount.
System checks balance.
IF amount<balance THEN Debit Amount ELSE Show Error.
N: Eject card.
Y: System dispenses cash.
System prepares printer.
System prints receipt.
Eject Card.
```

If-Then-Else sentences accept only one instruction and always have to contain Else. A sentence that starts with “Y:” will only execute whether the execution path enters the previous If statement. A sentence that starts with “N:” will only execute whether the execution path enters a previous Else statement. Synonym terms can be defined in a dictionary, *e.g.*, “Quantity” as a synonym of “Amount”.

Sentences from Listing 1 plus a dictionary with synonyms terms produce the Activity Diagram presented in Figure 3 and the test cases presented in Figure 4.³ Letters “Y” and “N” in the Figure 3 represent “Yes” and “No”, respectively. Although their Activity Diagram represents the activities “Start” and “End”, in some cases (paths) it did not represent an “End” activity or an arrow to an existing “End” activity, like in two activities named “Eject Card” in Figure 3. Test cases in the Figure 4 are produced from the possible execution paths. Likewise, test coverage criteria are based on path coverage.

² Example retrieved from Rane (2017), *op. cit.*, p. 47.

³ Both figures are extracted from Rane (2017), *op. cit.*, pages 49 and 50.

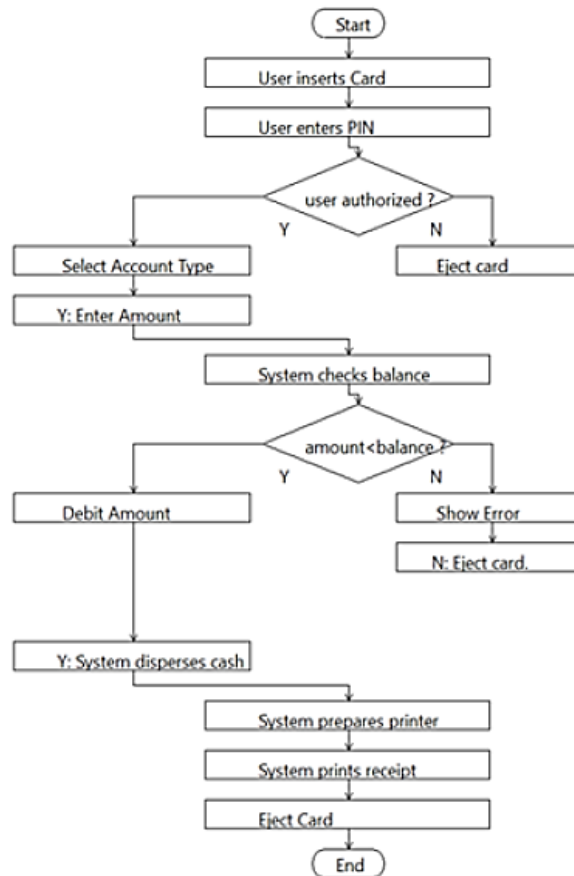


Figure 3 – An Activity Diagram produced by Rane's approach

Test Cases for ATM Cash Withdrawal	
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter positive value → System checks balance → amount<balance? → Debit Amount → Y: System dispenses cash → System prepares printer → System prints receipt → Eject Card → end	
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter positive value → System checks balance → amount<balance? → Show Error → N: Eject card → end	
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter negative value → System checks balance → amount<balance? → Debit Amount → Y: System dispenses cash → System prepares printer → System prints receipt → Eject Card → end	
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter negative value → System checks balance → amount<balance? → Show Error → N: Eject card → end	
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter zero value → System checks balance → amount<balance? → Debit Amount → Y: System dispenses cash → System prepares printer → System prints receipt → Eject Card → end	
User inserts Card → User enters PIN → user authorized? → Select Account Type → Y: Enter Amount → Enter zero value → System checks balance → amount<balance? → Show Error → N: Eject card → end	
User inserts Card → User enters PIN → user authorized? → Eject card → end	

Figure 4 – Example of test cases produced by Rane's work

Rane's work also presents the adopted NLP techniques – we detail them in the section 5.1 – and compares the time and the effort taken for generating test cases with the manual approach. According to its findings, the tool increases the time taken to generate test cases for a single feature by 7% but it reduces the effort by 31%. For multiple features, the time and effort are reduced by 61% and 87% respectively. Since the participants of the study had no prior experience with the tool, they expect that this time can reduce over time.

Although Rane's approach can generate test cases, it does not generate test data, test oracles, nor test scripts, and it cannot execute them. Since its input is not based on plain-text files, it is highly probable that the specification artifacts are not friendly to use with version control systems and, therefore, they have high maintenance costs, especially when concurrent modifications occur. The work does not address the static verification of the specification in detail. It only mentions performing some "error handling" such as verifying the format of user stories and checking for empty declarations.

Elghondakly *et al.* (2015) propose a requirement-based testing approach for automated test generation for Waterfall and Agile models. Their approach claims to parse functional and non-functional requirements for generating test paths and test cases. However, it does not discuss any implementation aspects such as techniques for parsing, or the format of user stories that are parsed, and it does not follow a model-based approach nor evaluate the coverage of the test cases. The generated test cases are not full-featured, *i.e.*, do not contain test data and oracles, and the approach does not generate test scripts. The authors do not offer details about the static verification of the specification.

Kamalakar *et al.* (2013) generate Java unit test scripts (JUnit) from features and scenarios written in Gherkin (section 4.4.2) for the English language. Class names are derived from feature names. Method names are derived from (camel-cased) scenario names. Parameters' data types from Given-When-Then sentences (see 4.1.2) are inferred to create method parameters. Assertions are created from Then sentences – string or numeric parameters are transformed into `assertEquals`; sen-

tences with a negative tone become `assertFalse`; otherwise, they become `assertTrue`. We detail the adopted NLP techniques in the section 5.1. The approach tries to facilitate the creation of unit level tests and, therefore, is not concerned with functional tests (that can verify high-level requirements) or with the validation of requirements with stakeholders. Static verification of the specification is not addressed by their work.

Verma & Beg (2013) propose the use of natural language to specify software requirements in the mathematics domain for generating test cases that explore the interval of numeric ranges using boundary value analysis. However, their work does not use Agile DSLs nor details the input format or exemplifies or details the output format (test cases). Thus, *we decided to not include it in our comparison*.

3.2. Research gaps

We identified many research gaps regarding the analyzed scope. In our opinion, the main reasons are twofold: (i) the use of NLP for processing agile requirements specifications has not been explored in depth; and (ii) researchers have not yet tried to adapt existing techniques for an agile development.

Research on V&V with use cases, for example, received more attention on the past decades. Although use cases can be used for agile development, both literature and industry have been affirmed (recently, although) that agile companies often adopt user stories and related DSLs for specification (CURCIO et al., 2018; DINGSØYR et al., 2012; INAYAT et al., 2015; SCHÖN; THOMASCHEWSKI & ESCALONA, 2017; STAVRU, 2014).

We enumerate ten important research gaps. Our approach tries to mitigate all of them:

1. *Static validation of requirements*: just a basic checking is proposed, such as validating if user stories follow their corresponding DSL. More verifications can help users to detect problems in specifications and correct them before they propagate to other artifacts;

2. *Support for more than one spoken language*: current approaches do not offer support for languages other than English, which reduces their broad application;
3. *Test cases and test scripts that combine different features and scenarios*: this a vital attribute for testing complex systems;
4. *Minimization and prioritization of generated test cases and test scripts*: approaches do not use information from requirements for reducing the number of test cases to generate – *e.g.*, to generate only test cases that cover error handling, to select only those scenarios tagged with an importance value higher than a certain number for generating test cases. They also do not apply any technique to reduce the number of test scripts to execute, *e.g.*, running only test scripts of the feature “x”. All these practices are important to making frequent tests feasible by reducing testing time;
5. *Generation of test cases with (relevant) test data*: Current approaches do not generate test cases with test data. More than just generating test data, these data need to try to reveal defects in the target applications;
6. *Generation of test data from external data sources*: To simulate or use real input data or to create tests that use data from existing systems, testers may need to access data from external data sources, such as test databases. This requires to provide means of integrating specifications and data sources;
7. *Generation of test cases with test oracles*: Oracles need to be aligned with the input data and the input actions. Their generation requires inferring the data to use, selecting the appropriate path through the test scenario (*i.e.*, the path that matches the expected behavior), and producing the corresponding verifiers;
8. *Generation of full-featured test cases as examples for requirements validation*: Current approaches only generate test cases that walk the specified scenarios without providing (new) data or oracles. To simulate different behaviors for validation with stakeholders, it may be necessary to create new examples;
9. *Generation of functional tests scripts*: Current approaches do not generate tests that verify whether an application complies with its requirements specifications;

10. *Execution and analysis of test scripts*: Automatic execution and analysis of test scripts are important for the test automation and, consequently, for reducing test costs and time.

3.3. Concluding remarks

This chapter analyzed and compared published works regarding the verification and the validation of requirements specified with natural language and Agile DSLs (chapter 4). The comparison used a set of 31 criteria that considered how the approaches deal with specifications, test cases, and test scripts. The approach presented by this thesis can fill most of their gaps about V&V (see Table 2).

4 Agile DSLs and Metalanguages

High thoughts must have high language.
- Aristophanes (450 BC-388BC), in *The Frogs*

This chapter presents DSLs commonly used in agile software development, related metalanguages, and discusses their adoption for test automation.

The increasing complexity of modern software systems instigates the need of raising the level of abstraction at which software is designed, implemented and tested. Domain-specific languages emerged in response to this need as an alternative to express software solutions in relevant domain concepts, thus hiding fine-grained implementation details and favoring the participation of domain experts in the software development process (JÉZÉQUEL et al., 2015).

In agile software development (ASD), DSLs have contributed to standardizing the way in which requirements are specified and used to create test scripts. Domain-specific languages for specifying requirements, like those presented in this chapter, facilitate discussion among the software team and between the team and stakeholders since they establish easy-to-read and simple-to-write textual patterns (ADZIC, 2011). Business analysts, for example, may spend less time thinking on *how* to write requirements and more on *what* they need to write. Software tools that integrate with these DSLs help developers and testers to create test scripts focused on the specified requirements (GÄRTNER, 2012). By amplifying collaboration among participants and reducing the loopback from requirements to coding and to testing, such DSLs may lead to an increase in the team's productivity (PUGH, 2011).

4.1.Common DSLs

User stories are the most frequently used artifact in agile software development (CURCIO et al., 2018; INAYAT et al., 2015; SCHÖN; THOMASCHEWSKI & ESCALONA, 2017). They present a low learning curve and have been successfully used to validate requirements with stakeholders (ADZIC, 2009, 2011; SMART, 2014; WYNNE & HELLESØY, 2012). Features documented as user stories often represent acceptance criteria as scenarios, written in a form of examples. The following subsections provide more details about the syntax commonly adopted by DSLs used to document user stories and scenarios. Anti-patterns and parameterization are also discussed.

4.1.1.User Story

Dan North (2006) introduced a DSL for a user story, aiming at helping companies to identify the business value of a feature and, hence, to prioritize features. The DSL addresses three fundamental elements of requirement engineering (WAUTELET et al., 2014): (i) *who* wants the functionality; (ii) *what* functionality stakeholders want the system to provide; and (iii) *why* stakeholders need the functionality. Listing 2 presents the DSL's template. The order of the sentences may vary without losing the meaning. There are also commonly accepted variations, such as "I want to" instead of "I would like to", and "So that" instead of "In order to". Listing 3 shows an example of a feature described with this DSL.

Listing 2 – DSL for User Story

```
As a <role or person>
I would like to <some feature>
In order to <benefit or added value>
```

Listing 3 – Example of a Feature

```
Feature: Add product to the shopping cart
As a visitor
I would like to add a product to my shopping cart
In order to buy it later
```

Our metalanguage adopts this DSL for describing features, but it does not use its sentences for test case generation.

4.1.2.Scenario

North (2006) also introduced a DSL for describing scenarios, aiming at helping to identify feature's acceptance criteria and at breaking its user story into verifiable fragments that can be checked by automated tests. Listing 4 presents the DSL's template. In case of having more than one precondition, action, or postcondition, each of them must be written in the next line and be preceded by the connector "And" – or eventually "But". Listing 5 shows an example of a scenario described with the DSL.

Listing 4 – DSL for a Scenario

```
Given <precondition or initial context>
When <action or event>
Then <postcondition or outcome>
```

Listing 5 - Example of a Scenario

```
Scenario: Add product by dragging and dropping
  Given that I have selected a product
  When I drag the product's image to the shopping cart icon
    And I drop it
  Then the product is added to the shopping cart
```

Smart (2014) observes that:

- the *Given* step should contain all the preconditions or steps that must have occurred *before* the actions of a scenario;
- the *When* step should contain actions or events in terms of *what* should happen, not *how*; and
- the *Then* step should describe the postconditions or outcomes expected from the scenario.

Our metalanguage uses Given-When-Then (GWT) steps in the following language constructions: Scenarios, Variants (section 6.1.11), Test Cases (section 6.1.12), and Test Events (section 6.1.13).

4.1.3. Anti-patterns

When developers not experienced with Agile DSLs start writing scenarios, they may not pay attention to their meaning or intent. Smart (2014) points out common anti-patterns (*i.e.*, practices not recommended), such as:

- a) *Scenarios steps that reflect technical implementation, instead of the business intent.* New practitioners may try to describe scenarios in terms of user interface interactions. Instead, they should describe what they are trying to perform or to achieve, from a business point of view. Otherwise, the specification will mix business and computing jargon, and business needs will be permeated with expected system behavior. **Our approach addresses this problem by making a clear separation of these concerns. Business needs are specified through Scenarios. Variants and Test Cases reflect the expected user-system interaction to comply with a Scenario.;**
- b) *Too long scenarios:* the length of a scenario may impact in its easiness to read, which reduces its using as a communication medium. **We believe that business-related scenarios are shorter than those that describe user-system interactions since they present a higher level of abstraction focused on the intent (“what”) instead of on the procedure (“how”). In Variants, which are used to describe user-system interactions, steps may refer to other features or scenarios by their produced states, instead of repeating the corresponding steps. Hence, this modularization may reduce the number of steps;**
- c) *A Given step that does not declare a verifiable precondition:* the sentence should not be vague about its precondition state, to enable its testing. **Our approach addresses this problem by letting a Variant to produce**

postconditions in its Then steps, as a consequence of its prior actions, and to refer to other Variant’s postconditions in Given and When steps. These references are verifiable (the states should match). Given steps without such preconditions can declare assertions about the state of the user interface;

- d) *A Then step that does not declare a verifiable expected outcome*: it should focus on a postcondition whose state can be verified by a stakeholder.

Our approach let postconditions and assertions be declared in Then steps of a Variant;

We propose the usage of Scenarios to specify expected behavior from a business point of view – the “*what*” part. Their sentences are not considered for generating test cases, but for discussing high-level business needs with stakeholders. We use Variants and Test Cases to define expected user-system interactions, corresponding to a Scenario – the “*how*” part. Section 6.1 presents details on these language constructions. Briefly, a Variant works like a higher-level template for generating Test Cases.

4.1.4.Parameterization

Some Scenarios, like those in Listing 6, can be very similar and vary only by some parameter values. These cases are candidates to become a **Parametrized Scenario** – *i.e.*, a kind of data-driven test.

Listing 7 shows an example of a Parameterized Scenario. Its values come from tabular data declared in the specification. These tabular data works like a table whose first row is the parameter name and the other rows are the values.

Listing 6 –Scenarios that vary by values

<p>Scenario: Receive a discount of 5% with coupon</p> <p>Given that I have selected the product Xpto</p> <p>When I enter a coupon named “OFF5”</p> <p>Then I receive a discount of 5 percent</p> <p>Scenario: Receive a discount of 10% with coupon</p>

```

Given that I have selected the product Xpto
When I enter a coupon named "OFF10"
Then I receive a discount of 10 percent

```

Listing 7 - Example of a Parameterized Scenario

Scenario: Receive a discount with coupon

```

Given that I have selected the product Xpto
When I enter a coupon named <name>
Then I receive a discount of <percent> %

```

Examples:

name	percent	
OFF5	5	
OFF10	10	

4.2. Specification of non-functional requirements

Some authors (AMBLER, 2008; COHN, 2004, 2009; DAVIES & SEDLEY, 2009) propose to represent non-functional requirements (NFR) as user stories or acceptance criteria. Listing 8 illustrates an NFR specified as a feature's user story. The feature contains a verifiable scenario, which serves as a testable acceptance criterion. Although simple, these examples can be verified by automated tests. When an NFR becomes difficult or even impossible to automate (*e.g.*, a usability or regulatory NFR) it can be checked through exploratory testing or inspections (AMBLER, 2008; LEFFINGWELL, 2011). Anyway, it is important to represent them. Inayat *et al.* (2015) point out that neglecting non-functional requirements is a challenge in agile requirements engineering.

Listing 8 - Example of NFR as a feature

```
Feature: Run in old OS versions
  In order to run the app in some old mobile phones
  As a customer
  I want to be able to run the app on Android 4 or above

Scenario: Run on Android 4
  Given that I installed the app on Android 4
  When I start it
  Then it opens without errors
  And all the automated functional tests pass
```

4.3.Integration with source code

A pattern adopted by tools that work with the aforementioned DSLs is using a regular expression to hook up a test method to a sentence from a requirement specification file. A sentence or *step* is transformed into a regular expression that matches exactly the given text and ignores any parameters or values. For example, the sentence “When I enter a coupon named <name>”, from Listing 7, will be transformed into “^When I enter a coupon named (.*)\$”.⁴ Depending on the testing framework, a *test skeleton* – often referred to as *step definition* – can be produced. Listing 9 shows an example of a step definition in Ruby and Listing 10 shows the same example in Java. Both can be produced by Cucumber (HELLESØY, 2009). The Given- and Then-steps are defined in a similar way.

Listing 9 – Step definition in Ruby

```
When /^When I enter a coupon named (.*)$/ do | name |
  # TO-DO: add code here
End
```

Listing 10 – Step definition in Java

```
@When("^When I enter a coupon named (.*)$")
public void I_enter_a_coupon_named(String name) {
  // TO-DO: add code here
}
```

⁴ The symbol “^” means “starting with”; the symbol “\$” means “ending with”; and “(.)” can be interpreted as “anything”.

Running such test methods will usually print the corresponding sentences, which produces the effect of seeing the specification to execute, step-by-step. When these test methods are filled with the code that corresponds to the sentence, they become an “*executable specification*”. In functional tests, a very common outcome is seeing the application under test being controlled through its GUI, while the sentences are printed. This lets both stakeholders and the development team monitor their compliance.

To reduce maintenance costs, text editors and IDEs (such as Visual Studio Code⁵, Atom⁶, Eclipse⁷, NetBeans⁸) offer plugins that can keep requirement files and test script files in sync – that is, a change in a *step* will update the corresponding *step definition*.

4.4. Metalanguages

BDD and ATDD books found in literature usually bring examples of features and scenarios written with the aforementioned DSLs. There are tools – like Cucumber⁹, JBehave¹⁰, and Behat¹¹ – that can integrate specifications written with these DSLs to test scripts (*i.e.*, source code). They support DSLs in different spoken languages (*e.g.*, English, Portuguese, French), although these DSLs contain the same language structures. These structures form a *metalanguage*. A lexer or parser written for such metalanguages need to use dictionaries to recognize sentences according to the target spoken language. Their syntax is often line-oriented and designed to be human-readable although non-technical.

We adopted three approaches to gathering these metalanguages. The first one consisted in searching for BDD and ATDD books¹² and then evaluating adopted

⁵ <https://code.visualstudio.com>

⁶ <https://atom.io>

⁷ <http://www.eclipse.org>

⁸ <https://netbeans.org>

⁹ <https://cucumber.io>

¹⁰ <http://jbehave.org>

¹¹ <http://behat.org>

¹² Search for books was conducted on Amazon (<https://www.amazon.com>) and Google Books (<https://books.google.com>), because of their abundance of computer science books.

frameworks and tools. The second was searching for BDD and ATDD frameworks and tools in source-code hosting services.¹³ Finally, using search engines.¹⁴ Keywords used (in all approaches) were: “BDD”, “ATDD”, “SbE”, “Behavior-Driven Development”, “Acceptance Test-Driven Development”, and “Specification by Example”. Titles and summaries were used as exclusion criteria. Metalanguages not used to specify requirements (*e.g.*, Galen¹⁵, KarateDSL¹⁶, EasyAccept¹⁷) or whose tools were not available for download were excluded as well.

4.4.1.JBehave

JBehave (NORTH, 2003) introduced the first metalanguage for BDD (WIKIPEDIA, 2017).¹⁸ It uses the concepts of “Story” and “Narrative” instead of “Feature”, with the same meaning and similar DSLs. It also adopts the concept of “Meta” for categorization and “GivenStories” for defining preconditions. Listing 11 shows an example in JBehave. In the example, the keyword `Meta` adds alternative identifications to the scenario name. These identifications make it easier to filter a scenario for execution or for combination with other user stories. The keyword `GivenStories` specifies one or more files whose scenarios are preconditions. It can be used in the context of the entire user story or individual scenarios. Whether the current user story or scenario does not need to depend on all the scenarios of a certain user story file, it may use an anchor to specify the scenarios from which it depends on, by their Meta (*e.g.*, “`GivenStories: path/to/select-product.story#{id1:scenario1;id2:scenario2}`”). A Meta can also be used to filter the scenarios to execute.

¹³ Used platforms were GitHub (<https://github.com>), GitLab (<https://gitlab.com>), and SourceForge (<https://sourceforge.net>), because of their current market share. Only the first 100 results of each keyword were considered.

¹⁴ Used search engines were Google (<https://www.google.com>) and Bing (<https://www.bing.com>), because of their current market share. Only the first 100 results of each keyword were considered.

¹⁵ <http://galenframework.com/>

¹⁶ <https://github.com/intuit/karate>

¹⁷ <http://easyaccept.sourceforge.net/>

¹⁸ Since the metalanguage did not receive a name, we refer to it as “JBehave”.

Listing 11 – Example in JBehave

```
Add product to the shopping cart

Narrative:
    As a visitor
    I would like to add a product to my shopping cart
    In order to buy it later

!-- A precondition to the entire user story
GivenStories: /path/to/select-product.story

Scenario: Add product by dragging and dropping
Meta: @id1 scenario1
    Given that I have selected a product
    When I drag the product's image to the shopping cart icon
    And I drop it
    Then the product is added to the shopping cart
```

4.4.2. Gherkin

Gherkin (HELLESØY, 2009) is probably the most common metalanguage used by tools that support BDD or ATDD. Currently, it supports over 70 spoken languages¹⁹, has syntax highlight supported by many text editors and IDEs (*e.g.*, Visual Studio Code, Sublime Text, Atom, TextMate, Vim, IntelliJ, Eclipse), and provides integration with programming languages like Python, Ruby, JavaScript, Java, Go, and Lua. Well known by being the language of Cucumber, it was heavily influenced by JBehave, although they evolve in parallel (WYNNE & HELLESØY, 2012). Listing 12 shows an example in Gherkin. Unlike JBehave, it does not have a way to specify preconditions or to refer other features or scenarios. Categories are denoted by the character “@” and can be used to filter the scenarios to execute. In order to avoid the repetition of steps in scenarios, it offers the construction “Background”, which is necessarily executed before every scenario. Background’s sentences use the same DSL as Scenarios.

¹⁹ <https://docs.cucumber.io/gherkin/reference/#spoken-languages>

Listing 12 – Example in Gherkin

```

Feature: Add product to the shopping cart
  As a visitor
    I would like to add a product to my shopping cart
    In order to buy it later

@important
Scenario: Add product by dragging and dropping
  Given that I have selected a product
  When I drag the product's image to the shopping cart icon
    And I drop it
  Then the product is added to the shopping cart

```

4.4.3. Robot

Robot Framework's metalanguage (NOKIA CORPORATION, 2008) – referenced here as “Robot” – is based on the Laukkanen's thesis (2006). The thesis compares data-driven and keyword-driven testing techniques and proposes a framework concept for future implementation. The metalanguage – produced along with the framework some months after the thesis – adopts a markup syntax and a keyword-driven approach with a tabular test data. Markup syntax is based on reStructuredText (GOODGER, 2002), a textual format created for writing technical documentation and simple web pages. A specification written in Robot must use sections, such as “Setting”, “Variables”, “Test Cases”, and “Keywords” to separate its content. Test cases' sentences can be declared with the Given-When-Then format, although is not required. Their corresponding behavior is defined in section “Keywords”. Fixed high-level commands that resemble natural language can be written using a tabular format. Listing 13 shows an example in Robot.

Listing 13 – Example in Robot

```

*** Settings ***
Documentation      Add product to the shopping cart
...
...               A visitor must be able to add a product to its
...               shopping cart in order to buy it later

Resource          select-product.robot

*** Test Cases ***
Add product by dragging and dropping
    Given that I have selected a product
    When I drag the product's image to the shopping cart icon
    and I drop it
    Then the product is added to the shopping cart

*** Keywords ***
# The keyword "That I have selected a product" is declared in
# the file select-product.robot

I drag the product's image to the shopping cart icon
    Drag And Drop    #product-1-img    #shopping-cart

I drop it

The product is added to the shopping cart
    ${cookie}=      Get Cookie
    Should Be Equal  ${cookie.name}    Cart-Product-1

```

4.4.4. Gauge

Gauge (THOUGHTWORKS, 2014) is based on the Markdown format (GRUBER, 2004)²⁰, which has been supported by source-code hosting services (e.g., GitHub, GitLab, SourceForge, BitBucket) and Wiki-based tools.

²⁰ Markdown was standardized by IANA in 2016 under the RFC 7763.

Listing 14 – Example in Gauge

```
# Add product to the shopping cart  
  A visitor must be able to add a product to its shopping cart in  
  order to buy it later  
  
## Add product by dragging and dropping  
  * A product must be selected  
  * Drag the product's image to the shopping cart icon  
  * Drop it  
  * The product must be added to the shopping cart
```

4.4.5. Comparison

Table 3 compares the metalanguages and also includes Concordia – the metalanguage introduced in this thesis – for helping to recognize similarities and differences. A full syntax or approach comparison is out of the scope of this thesis, due to the large number of details.

Table 3 - Comparison of Metalanguages

#	Group	Item	<i>Gherkin</i>	<i>JBehave</i>	<i>Robot</i>	<i>Gauge</i>	<i>Concordia</i>
1	Info.	<i>Website</i>	cucumber.io	jbehave.org	robotframework.org	gauge.org	concordialang.org
2		<i>Open-sourced license?</i>	Yes, MIT	Yes, BSD	Yes, Apache	Yes, GPL	Yes, AGPL
3		<i>First release year</i>	2011	2003	2008	2014	2018
4		<i>Created by a company?</i>	No	No	Yes, Nokia	Yes, Thoughtworks	No
5		<i>Sponsored by a company?</i>	Yes, Cucumber Ltd	No	Yes, Robot Framework Foundation	Yes, Thoughtworks	No
6		<i>Original programming language</i>	Ruby	Java	Python	Go	JavaScript
7		<i>Spec. file extensions</i>	.feature	.story	.robot	.spec, .md, .cpt	.feature, .testcase
8		<i>Plugins for test generation</i>	14+ ²¹	1	2	6	2

²¹ <https://cucumber.io/docs#cucumber-implementations>

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
9		<i>Supported application platforms</i>	web, mobile native, mobile hybrid, mobile web, desktop	web	web, java GUI	web	web, mobile native, mobile hybrid, mobile web, desktop
10	Approach	<i>Translatable?</i>	Yes	Yes	No ²²	Not Applicable ²³	Yes
11		<i>Context-free grammar available?</i>	Yes ²⁴	Yes ²⁵	No, but there is a well-defined syntax ²⁶	No, just examples ²⁷	Yes, see Appendix B
12		<i>Data-driven testing?</i>	Yes	Yes	Yes	Yes	Yes

²² Internationalization support was cancelled (<https://github.com/robotframework/robotframework/issues/2282>). Given-When-Then syntax in other languages than English was not implemented since 2014 (<https://github.com/robotframework/robotframework/issues/519>). SeleniumLibrary, which is used by Robot Framework, did not translate its commands yet.

²³ Gauge adopts Markdown which uses symbols instead of keywords.

²⁴ <https://github.com/cucumber/cucumber/blob/master/gherkin/gherkin.berp>

²⁵ <http://jbehave.org/reference/stable/grammar.html>

²⁶ <http://docutils.sourceforge.net/docs/ref/rst/roles.html>

²⁷ <https://daringfireball.net/projects/markdown/syntax>

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
13		<i>Can data-driven tests use external data sources?</i>	No	No	Yes, Robot and TSV files	Yes, CSV files	Yes, CSV, Excel, Ini, and JSON files; Access, Firebase, MySQL, SQLServer, PostgreSQL, and SQLite databases
14		<i>Can it filter data from external data sources?</i>	No	No	No	No	Yes, using SQL, even for files
15		<i>Keyword-based testing?</i>	No	No	Yes	No	No, but it uses State-based testing, that works in a similar way. ²⁸
16		<i>Can it declare test cases?</i>	Yes, but as scenarios	Yes, but as scenarios	Yes	Yes, but as scenarios	Yes

²⁸ Concordia replaces required states by their producers' steps, when it generates test cases. Keyword-based testing approaches replaces keywords by their corresponding definition, when they generate test scripts.

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
17	Usage for Testing	Can integrate with test scripts?	Yes	Yes	Yes	Yes	Yes
18		Can be used to generate test scripts skeletons?	Yes	Yes	Yes	Yes	Yes, with Gherkin-based tools
19		Can generate test data automatically?	No	No	No	No	Yes
		Can generate test oracles automatically?	No	No	No	No	Yes
		Can combine scenarios or test cases automatically?	No	No	No	No	Yes
		Can generate test cases automatically?	No	No	No	No	Yes
21	DSL	Language	Yes, with a special comment	No	No	Not Applicable	Yes, with a special comment

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
22		<i>Import declarations</i>	No	Yes, but only as preconditions, with “GivenStories”	Yes, with “Resources”	No	Yes, with “Import”
23		<i>Feature</i>	Yes, with “Feature”	Yes, as the first line content or using “Feature”	No	Yes, with a symbol (#)	Yes, with “Feature”
24		<i>Feature description</i>	AIS	AIS	No	free	AIS
25		<i>Scenario</i>	Yes, with “Scenario”	Yes, with “Scenario”	No	Yes, with a subsection header	Yes, with “Scenario”
26		<i>Scenario description</i>	GWT	GWT	N/A	Using bullets	GWT
27		<i>Shared base scenario</i>	Yes, with “Background Scenario”	Yes, with “Background Scenario”	No	Yes, with Context or Context Steps	Yes, with “Background Scenario”
28		<i>Parameterized scenario description</i>	Yes	Yes	Yes	Yes	GWT
29		<i>Categorization</i>	Yes, with tags	Yes, with tags and meta	Yes, with meta	Yes, with tags	Yes, with tags

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
30		Table	Yes	Yes	No, but can simulate it by combining tabulated parameters with for loops	Yes	Yes
31		Test case	No	No	Yes	No	Yes
32		Test case description	N/A	N/A	free, but GWT is recommended	N/A	GWT
33		Test case events	No	Yes: “Lifecycle” with “Before” or “After” and “Scope” with “STORY” or “SCENARIO”	Yes: “Suite Setup”, “Suite Tear Down”, “Test Setup”, “Test Tear Down”	Yes: using contexts as setup, and “___” (three underscores) as tear down	Yes: “Before All”, “After All”, “Before Feature”, “After Feature”, “Before Each Scenario”, “After Each Scenario”
34		Variable or constant	No	No	Yes, both ²⁹	No	Yes, Constants
35		Data source	No	No	Yes, with “file”	Yes, with “file” or “table”	Yes, with “Database”

²⁹ Robot has Constants and Variables, but Constants are accessed through Variables.

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
36		User interface element	No	No	No	No	Yes, with “UI Element”
37	DSL capabilities	Avoids duplication of steps in Scenarios from the same Feature	Yes, with “Background”	Yes, with “Background”	Yes, with “Keywords”	Yes, with Contexts or Context steps	Yes, with “Background”
38		Allow to include Scenarios or steps from other Features	No	Yes, with “GivenStories”	Yes, with “Resources”	No	Yes, with states
39		Tags can be used to identify related cross-functional concerns	Yes	Yes	Yes	Yes	Yes
40		Has reserved categories (tags/meta), with some special meaning	Yes, but it varies according to the used tool (e.g., @issue)	Yes: @issue, @tag, @tags	No	No	Yes, see 6.1.4

#	Group	Item	Gherkin	JBehave	Robot	Gauge	Concordia
41		<i>Uses tags for referencing other declarations</i>	No	No	No	No	Yes, See 6.1.4
42		<i>Uses tags for filtering what to execute</i>	Yes	Yes	Yes	Yes	Yes
43		<i>Uses tags for prioritization</i>	Yes	No	No	No	Yes, see 7.3.2

All the approaches utilize data-driven testing, which enforces its adoption. Likewise, all but Gauge use GWT syntax for scenarios or test cases (although it is optional in Robot). Robot, however, does not offer explicit support to features and scenarios.

Gherkin, JBehave, and Gauge do not offer an explicit syntax for test cases. Since test cases have to be declared as scenarios, the requirements specification may end up mixing high-level and low-level scenarios. This mix makes it harder to read and maintain and may affect the validation with stakeholders (*e.g.*, to find only high-level scenarios for discussion). Besides, these three metalanguages cannot be used to convert low-level sentences into source code. Thus, developers or testers need to codify them manually, which increases costs and schedule.

Robot does not support other languages than English, which probably limits its adoption. Like other keyword-based approaches, its syntax may offer higher learning curve (ISO; IEC & IEEE, 2016). Supported commands need to be written exactly as defined, and the tabulated syntax mixed with symbols refers to spreadsheets or to programming, instead of a natural language specification – which makes the specification harder to read and understand. Concordia, on the other hand, tries to mitigate this problem by adopting a syntax based on restricted natural language.

Robot's keywords can be reused in different test cases, but it may be necessary to zigzag through the document (or documents) to understand all the context or sequence of operations. A sequential declaration is possibly easier to validate with stakeholders. In Concordia, the sentences of a Variant – *i.e.*, a lower-level scenario that establishes the expected interaction with the UI and serves as a template to generate test cases – are declared sequentially. Listing 15 presents an example that contains a Variant.

References to external content are represented differently by the metalanguages. Gherkin does not have such references. Gauge uses links but does not validate them. JBehave allows references to other feature files using “GivenStories” (which is de-

clared to define preconditions in features or scenarios) and to external test data (currently only CSV files) used by data-driven tests. Robot allows references to keywords, test data and source code from other files, declared as resources. Concordia allows references to external constants, states, user interface elements, tables, and databases (all verifiable). States in Concordia are used to combine Variants and produce Test Cases. For example, the sentence “Given that I have ~product selected~” from Listing 15 has a reference to the state “product selected”, which is produced by a Variant of the imported feature file. Whether the feature file has more than one Variant with the referenced state, all of them can be combined with the current Variant to form different Test Cases (section 7.3.4 has more details).

All the metalanguages can be used to produce test skeletons (Concordia can be used by Gherkin-based tools since they are compatible). Robot and Concordia can be used to generate test scripts from the specification.

Listing 15 – Example in Concordia

```
import "select-product.feature"

Feature: Add product to the shopping cart
  As a visitor
    I would like to add a product to my shopping cart
    In order to buy it later

@important
Scenario: Add product by dragging and dropping
  Given that I have selected a product
  When I drag the product's image to the shopping cart icon
    And I drop it
  Then the product is added to the shopping cart

Variant: Produces a cookie
  Given that I have ~product selected~
  When I drag <#product-1-img> to <#shopping-cart>
  Then I see the cookie "Cart-Product-1"
```

4.5. Concluding remarks

This chapter presented an overview of domain-specific languages commonly used in ASD and compared the metalanguages created around similar concepts. The metalanguage proposed by this thesis considered recurring practices and concepts – aiming at trying to keep a low learning curve – and to improve their usage for V&V.

5Restricted Natural Language Processing

The limits of my language means the limits of my world.

- Ludwig Wittgenstein (Austrian philosopher)

This chapter briefly discusses some natural language processing (NLP) techniques and their using in the thesis' approach.

During our research, we investigated ways of recognizing natural language sentences and of understanding their meaning. A promising approach that we came across was *intent recognition* (IR), also known as *plan recognition*. We observed that many current solutions to construct *chatbots* (or chatterbots), *i.e.*, computer programs that try to simulate a human being in a conversation, are using IR (section 5.4 presents some NLP solutions that use IR) and it could fit our approach's needs. We then decided to investigate a little further whether we could use it to recognize the intent of some sentences, in relation to the desired actions, given values, etc. After preparing a small experiment to recognize Given-When-Then sentences' intent and of having successful results, we decided to try it with our approach. To the best of our knowledge, IR has not yet been used as the basis for recognizing Agile DSLs. We also could not find its use with Use Cases or other artifacts for documenting requirements.

5.1.Related work

We analyzed approaches that apply NLP to agile specifications as a mean of extracting conceptual models or of generating test cases. Our intention was to determine useful techniques that we could eventually use with our approach.

Rane (2017) produces test cases in English language and Activity Diagrams from features and scenarios written in Gherkin (section 4.4.2), for the English lan-

guage. The approach also uses a dictionary as input. This dictionary stores keywords commonly used in user stories and their associated steps. User input is performed through a graphical interface and features and scenarios are stored in a database. Adopted NLP techniques include *lemmatization* (section 5.2.2), *part-of-speech* (POS) *tagging* (section 5.2.1), *dependency parsing* (section 5.2.3), and *synonym generation*. The work uses the Stanford NLP library (DE MARNEFFE & MANNING, 2008) and WordNet lexical database (MILLER, 1995).

Robber *et al.* (2016) extract OWL ontologies from user stories written with the same DSL as the presented in section 4.1.1. Their process consists of parsing the user stories into tokens, applying *part-of-speech* (POS) *tagging* (section 5.2.1), inferring concepts and relationships to determine token weights, removing stop words from the collection of tokens, attaching a weight to each term (based on the frequency and on the weights that were specified as input parameters), and constructing a conceptual model from the weighted terms. The conceptual model – expressed as OWL ontologies – is transformed into a graphical representation and then presented to stakeholders for helping them to visualize dependencies and relationships between user stories and unnecessary or redundant roles.

Kamalakar (2013) derives unit test scripts (specifically, tests for Java classes) from features and scenarios written in Gherkin (section 4.4.2), for the English language. His approach consists of parsing features and scenarios using *regular expressions*; performing *lemmatization* (section 5.2.2), applying *part-of-speech* (POS) *tagging* (section 5.2.1), extracting quoted parameters from Given-When-Then sentences (see parameterization in section 4.1.4), using a probabilistic matcher to extract words, and generating source code from these words and parameters. Class names are derived from feature names. Method names are derived from (camel-cased) scenario names. Parameters' data types from Given-When-Then sentences are inferred to create method parameters. Assertions are created from Then sentences – string or numeric parameters are transformed into `assertEquals`; sentences with a negative tone become `assertFalse`; otherwise, they become `assertTrue`. The work also uses the Stanford NLP library and WordNet lexical database.

Soeken *et al.* (2012) proposes an assisted flow for BDD where the user enters into a dialog with the computer which suggests code pieces extracted from the sentences. Figure 5 shows an example of such piece of code. The work also uses the Stanford NLP library and WordNet lexical database. Figure 6 shows an example of a phrase three structure (Figure 6a) created with the Stanford Parser – which is part of the Stanford NLP library – and a list of typed dependencies for the sentence (Figure 6b). We detail these techniques (POS tagging and dependency parsing) in 5.2.1 and 5.2.3, respectively.

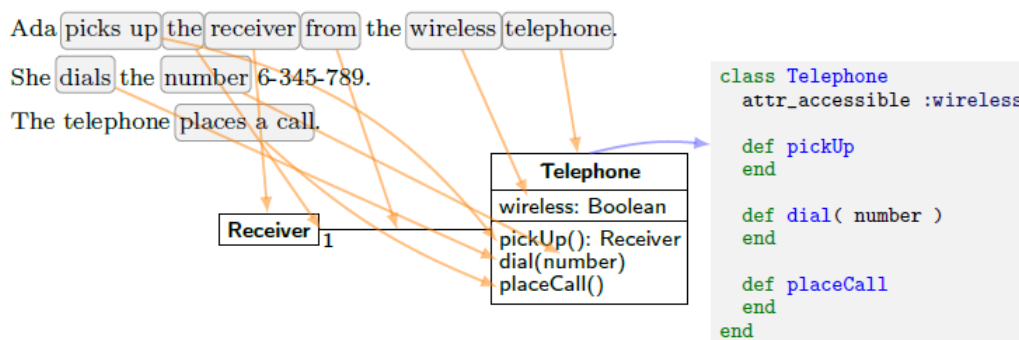


Figure 5 - Example of a Class Diagram generated by Soeken et al.'s work³⁰

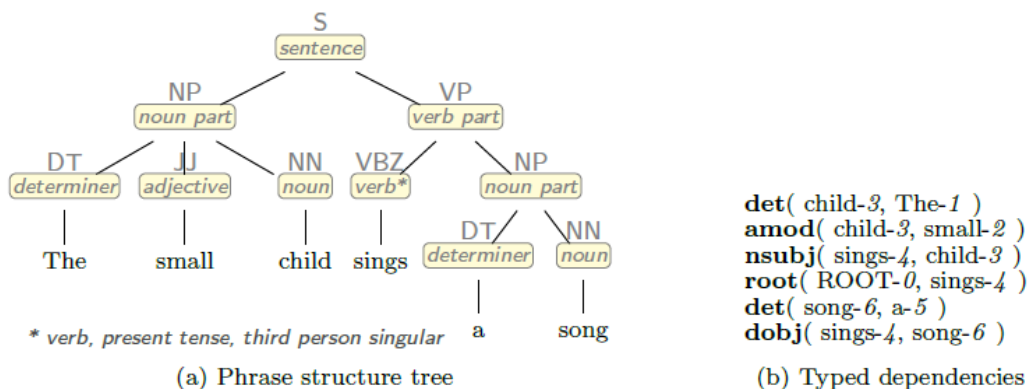


Figure 6 – Example of an application of the Stanford Parser³¹

Our approach uses stemming, context-free grammar, and intent recognition, through the adopted solution for NLP (section 5.2).

³⁰ Figure retrieved from Soeken *et al.* (2012), Figure 5.

³¹ Figure retrieved from Soeken *et al.* (2012), Figure 2.

5.2. Techniques

This section describes some common NLP techniques. They are used as building blocks by many approaches – like those in the previous section – and NLP solutions – like those from section 5.4.

5.2.1. Part-of-speech tagging

Part-of-speech (POS) tagging classifies each word of a speech lexically, as a noun, verb, adjective, adverbs, etc. This classification usually adopts a database of words – called *treebank* – organized in a tree form annotated with syntactic information. Every tag receives an identification. Table 4 shows the tags of the Penn Treebank (SANTORINI, 1990), used by some NLP solutions, such as the Stanford Parser (DE MARNEFFE & MANNING, 2008). The Penn Treebank is a human-annotated collection of 4.5 million words (MARCUS; SANTORINI & MARCINKIEWICZ, 1993) which groups elements with POS tags and *phrase tags* (*i.e.*, NP for noun phrase, PP for prepositional phrase, VP for verb phrase, or ADVP for adverb phrase). Phrase tags are assigned to a group of co-located words in a phrase.

Table 4 - Tags in Penn Treebank

Id	Description
CC	Conjunction, coordinating
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Conjunction, subordinating or preposition
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Verb, modal auxiliary
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Noun, proper singular
NNPS	Noun, proper plural
PDT	Predeterminer
POS	Possessive ending
PRP	Pronoun, personal
PRPS	Pronoun, possessive

Id	Description
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Adverb, particle
SYM	Symbol
TO	Infinitival to
UH	Interjection
VB	Verb, base form
VBZ	Verb, 3 rd person singular present
VBP	Verb, non-3 rd person singular present
VBD	Verb, past tense
VBN	Verb, past participle
VBG	Verb, gerund or present participle
WDT	<i>wh</i> -determiner
WP	<i>wh</i> -pronoun, personal
WPS	<i>wh</i> -pronoun, possessive
WRB	<i>wh</i> -adverb

To illustrate, the phrase “Concordia is a new metalanguage” would be tagged as “NPP VBZ DT JJ NN”, according to Table 4.

5.2.2. Stemming and lemmatization

The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form (MANNING; RAGHAVAN & SCHÜTZE, 2008). For instance, “am”, “are”, and “is” are reduced to “be”; “car”, “cars”, “car’s”, and “cars’” are reduced to “car”; applying the reduction to the sentence “the boy’s cars are different colors” results in “the boy car be differ color”.

Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes (MANNING; RAGHAVAN & SCHÜTZE, 2008). *Lemmatization* usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the *lemma* (MANNING; RAGHAVAN & SCHÜTZE, 2008). For instance, if confronted with the token “saw”, stemming might return just “s”,

whereas lemmatization would attempt to return either “see” or “saw” depending on whether the use of the token was as a verb or a noun. Stemmers use language-specific rules, but they require less knowledge than a lemmatizer, which needs a complete vocabulary and morphological analysis to correctly lemmatize words. Particular domains may also require special stemming rules. The most common algorithm for stemming English, and one that has repeatedly been shown to be empirically very effective (MANNING; RAGHAVAN & SCHÜTZE, 2008), is Porter's algorithm (PORTER, 1980).

NLP solutions presented in section 5.4 use stemming or lemmatization in their process to recognize entities or intents.

5.2.3. Dependency parsing

A dependency parser analyzes the grammatical structure of a sentence to indicate its subject, objects, eventual verb phrase between subject and an object, conditional statements, etc. Analysis' output often assumes the form of a tree, where each word is a node and a *root node* is head of the entire structure. Figure 7 illustrates such output.³² Relations among the words are labeled with direct arcs from heads to dependents. These labels usually receive dependency relations from the Universal Dependency set (DE MARNEFFE et al., 2014). Table 5 shows some of these relations.³³

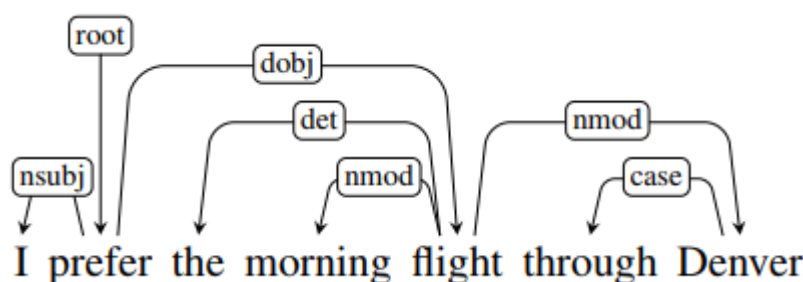


Figure 7 – Example of a dependency analysis

³² Figure retrieved from the book by Jurafsky & Martin (JURAFSKY & MARTIN, 2009), chapter 14, figure 14.1.

³³ Table retrieved from the book by Jurafsky & Martin (JURAFSKY & MARTIN, 2009), chapter 14, figure 14.2.

Table 5 - Selected relations from the Universal Dependency set

Causal Argument Relation	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relation	Description
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions, and other case markers
Other Notable Relation	Description
CONJ	Conjunct
CC	Coordinating conjunction

A major advantage of dependency grammars is the ability to deal with languages that are morphologically rich and have a relatively *free word order* (JURAFSKY & MARTIN, 2009) – that is, it abstracts away from word-order information and uses links to represent relationships.

5.2.4.Entity recognition

Entity recognition (ER) is the task of detecting entities in a sentence. An *entity* is an object or set of objects in the world. A *mention* is a reference to an entity. Entities may be referenced in a text by their name, indicated by a common noun or noun phrase, or represented by a pronoun. *Named entity recognition* (NER) is the task of recognizing entities by their name. For example, the following are several mentions of a single entity:

- **Name mention:** John Doe
- **Nominal mention:** The guy wearing a black shirt.

- **Pronoun mention:** he, him

Common entities types include: *quantity*, *currency*, *time*, *weather*, *location* (geographical areas, landmasses, bodies of water, geological formations), *person* (individual or group), *organization* (corporations and agencies), *geopolitical entity* (nation, region, government or people), *facility* (buildings, permanent man-made structures), and *place*. All these types may have subtypes (*e.g.*, the type “person” may have a subtype “religious figures”).

To illustrate, the phrase “U.N. official John Doe heads for Iraq” contains the organization “U.N.”, the person “John Doe”, and the location “Iraq”, and would receive tags corresponding to each type. Entity detection depends on *entity databases* for the desired types and subtypes (SANG & DE MEULDER, 2003).

5.2.5.Intent recognition

Intent recognition approaches are ultimately based on psychological and neuroscientific evidence for a theory of mind (PREMACK & WOODRUFF, 1978), which suggests that the ease with which humans recognize the intentions of others is the result of an innate mechanism for representing, interpreting, and predicting other’s actions (KELLEY et al., 2012). The mechanism relies on taking the perspective of others (GOPNIK; SLAUGHTER & MELTZOFF, 1994), which allows humans to correctly infer intentions.

Recognizing the intent of a textual sentence (*e.g.*, an utterance) is a difficult task (TAYLOR & MAZLACK, 2005). Sentences can have literal or non-literal meaning. For example, the literal meaning of “Can you close the door?” is “Are you capable of closing the door?”, while the indirect meaning is “Please close it”. People can learn through experience the indirect meaning of such things. Computational models can learn through examples and machine learning algorithms.

IR may involve stemming, POS tagging, dependency parsing, entity recognition, and machine learning (ML) approaches. The next section presents some ML approaches to natural language processing that can be used for intent recognition. The

usage of IR for creating chatbots is well discussed in the books by Jurafsky & Martin (JURAFSKY & MARTIN, 2009).³⁴ They classify chatbots in *rule-based* chatbots and *corpus-based* chatbots. The former uses rules to map user sentences into system responses. The latter mines logs of human conversation to learn to automatically map user sentences into system responses.

5.3.Approaches

Rule-based methods can be implemented with semantic grammars. A *semantic grammar* is a context-free grammar (CFG) in which the left-hand side of each rule corresponds to the semantic entities being expressed, *i.e.*, the *slot* names, (JURAFSKY & MARTIN, 2009) as in the fragment from Listing 16.

Listing 16 – Example of a semantic grammar³⁵

SHOW	→ show me i want can i see ...
DEPART TIME RANGE	→ (after around before) HOUR morning afternoon evening
HOUR	→ one two three four... twelve (AMPM)
FLIGHTS	→ (a) flight flights
AMPM	→ am pm
ORIGIN	→ from CITY
DESTINATION	→ to CITY
CITY	→ Boston San Francisco Denver Washington

Semantic grammars can be parsed with regular expressions or CFG algorithms. Figure 8 shows an example of a semantic grammar parse for a user sentence, using slot names as the internal parse tree nodes.

³⁴ A draft of the third version of the book is available at <https://web.stanford.edu/~jurafsky/slp3/>

³⁵ Retrieved from the book by Jurafsky & Martin (JURAFSKY & MARTIN, 2009), op. cit., chapter 29.

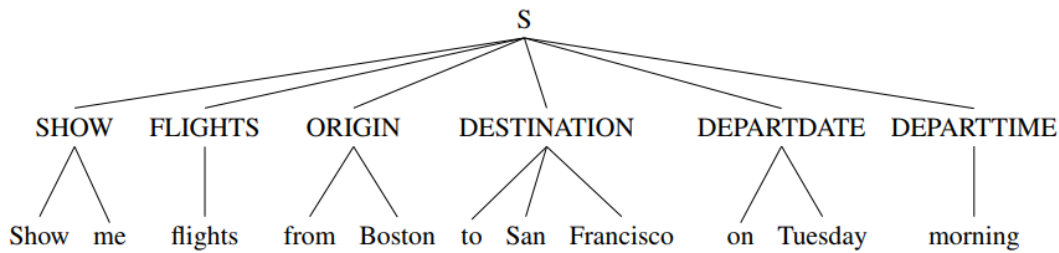


Figure 8 - Example of a semantic grammar parse³⁶

Rule-based approaches are very common in industrial applications (JURAFSKY & MARTIN, 2009). It has the advantage of *high precision*, and if the domain is narrow enough and experts are available, can provide sufficient coverage as well. On the other hand, the hand-written rules or grammars can be both expensive and slow to create, and hand-written rules can have *low recall* when it is difficult to predict all the possible variations for a slot.

A common alternative to rule-based approach is to use **supervised machine learning**. Assuming a *training set* is available which associates each sentence with the correct semantics, a classifier can be trained to map from sentences to intents and domains, and a sequence model to map from sentences to slot fillers. For example, a classifier (*i.e.*, neural network, naïve bayes, logistic regression method, etc.) can be applied to the sentence “I want to fly to San Francisco on Monday, please” to determine that the intent is “SHOW_FLIGHT”, the domain is “AIRLINE”, the destination is “San Francisco”, and the day of the week is “Monday”. Intent recognition strategies adopt techniques of information extraction such as entity recognition (ER) and named entity recognition (NER) – see section 5.2.4. In the example, a list of airports, cities, and days of the week are used by the classifier to establish the proper slot names. Capitalized words can be used as *markups* in the training corpus to help identifying entities.

Popular probabilistic models for supervised machine learning include *Naïve Bayes Classifier* (NBC), *Supporting Vector Machines* (SVM), *Averaged Perceptron* (AP), the *Conditional Random Field* (CRF) model, the *Maximum Entropy Model*

³⁶ Retrieved from the book by Jurafsky & Martin (JURAFSKY & MARTIN, 2009), op. cit., chapter 29.

(MEM), the *Hidden Markov Model* (HMM), and the *Maximum Entropy Markov Model* (MEMM) – *e.g.*, (CHEN et al., 2009a), (MEYER, 2001), (COLLINS, 2002), (LAFFERTY; MCCALLUM & PEREIRA, 2001), (ADWAIT RATNAPARKHI, 1996), (CUTTING et al., 1992), (MCCALLUM; FREITAG & PEREIRA, 2000) respectively. Different algorithms can be applied to these models, with applications for parsing, POS tagging, Intent Recognition, and biological sequencing (*e.g.*, identification of regions from genomic DNA). For **unsupervised learning** algorithms, we recommend seeing those summarized by Christodoulopoulos & Steedman (2010).

5.4. Solutions for Natural Language Processing

During our research, we decided to investigate solutions that were available to use – that is, solutions that had libraries, frameworks or web services – and their approaches. Table 6 presents the main solutions found,³⁷ grouped by type (column “Type”). They were collected using web search engines³⁸, opensource software hosting platforms,³⁹ and some reputable Q&A websites.⁴⁰ All of them support the English language. Since we opted to conduct case studies with Brazilian companies that specify requirements in Portuguese, we evaluated their support to Brazilian Portuguese (Pt-Br) and Portuguese (Pt). Their support for IR was evaluated as well. Some solutions were only available through web services and offered free plans with limited access (*e.g.*, a maximum number of requests).

We conducted small experiments with most solutions in Table 6 to evaluate whether we could use them to recognize sentences that expressed interactions with a user interface. Their capacity to recognize sentences varied a little, depending on the number of samples used for training or the sentences’ structures – although, we did not perform a formal evaluation in this matter. **In our opinion, Bravey, API.AI, Wit.AI, and Google NLP, were the friendliest approaches – *i.e.*, they were easier to setup and use – and they presented very good results with little training.**

³⁷ Until June 2017

³⁸ We searched with the most used web engines, according to Statista (www.statista.com) and Netmarketshare (www.netmarketshare.com), *i.e.*, Google (www.google.com), Bing (www.bing.com), and Yahoo (www.yahoo.com).

³⁹ GitHub (www.github.com), SourceForge (www.sourceforge.com), and GitLab (www.gitlab.com)

⁴⁰ StackOverflow (stackoverflow.com), Reddit (www.reddit.com), and Quora (www.quora.com)

We decided to use Bravey in our prototype tool for the following reasons:

- (i) it offers good results with little training;
- (ii) it supports Pt and Pt-Br;
- (iii) it supports Intent Recognition;
- (iv) it is a local, offline solution;
- (v) it is more concise and easier to adapt than other local solutions;
- (vi) it is opensource and can be adapted freely.

Table 6 – Solutions for Natural Language Processing

Name	License	Type	Limit	Prog. Lang.	Approaches	Pt-Br	Pt	I.R.
Algorithmia ⁴¹	free plan	web service	5k requests	many	N/A	Yes	Yes	Yes
API.AI ⁴²	free plan	web service	per hour	many	N/A	Partial	Yes	Yes
Google NLP ⁴³	free plan	web service	5k requests	many	N/A	No	Beta	Yes
MeaningCloud ⁴⁴	free plan	web service	40k req/month	many	N/A	No	Yes	Yes
Microsoft LUIS ⁴⁵	free plan	web service	10k req/month	many	N/A	Yes	No	Yes
Wit.AI ⁴⁶	free plan	web service	per hour	many	N/A	No	Yes	Yes
Adapt ⁴⁷	open	Local	No	Python	CFG	No	No	Yes
Apache OpenNLP ⁴⁸	open	Local	No	Java	MEM, AP, NBC	No	Yes	No
Apache Stanbol ⁴⁹	open	Local	No	Java	MEM, AP, NBC	No	Yes	No
Bravey ⁵⁰	open	Local	No	JavaScript	NBC, CFG	Yes	Yes	Yes
NLTK ⁵¹	open	Local	No	Python	NBC	No	Yes	No
RASA.AI ⁵²	open	Local	No	Python	SVM, CRF, AP, CFG	No	No	Yes
Stanford NLP ⁵³	open	Local	No	Java	CRF, CFG	No	Partial	No

⁴¹ <https://algorithmia.com/>⁴² <https://docs.api.ai/docs/languages>⁴³ <https://cloud.google.com/natural-language/docs/languages>⁴⁴ <https://www.meaningcloud.com/products/pricing>⁴⁵ <https://docs.microsoft.com/en-us/azure/cognitive-services/luis/luis-concept-language-support>⁴⁶ <https://wit.ai/blog/2016/04/28/new-languages>⁴⁷ <https://adapt.mycroft.ai/>⁴⁸ <https://opennlp.apache.org/>⁴⁹ <https://stanbol.apache.org/docs/trunk/components/enhancer/nlp/#supported-languages>⁵⁰ <https://github.com/BraveyJS/Bravey>⁵¹ <http://www.nltk.org/>⁵² <https://rasa-nlu.readthedocs.io/en/latest/languages.html#adding-a-new-language>⁵³ <https://stanfordnlp.github.io/CoreNLP/human-languages.html>

5.5.Intent recognition with Bravey

Bravey (VIDIEMME CONSULTING, 2016) is an opensource solution that offers a simple API to create conversational interfaces, like those used in chatbots. Its approach uses Naïve-Bayes Classifiers and provides customizable entity recognizers based on regular expressions chaining.

A Naïve-Bayes Classifier is a probabilistic model that uses a collection of labeled training examples to estimate the parameters of the given model. Classification on new examples is performed with Bayes' probability theorem by selecting the class that is most likely to have generated the example (MCCALLUM & NIGAM, 1998). It assumes that all the attributes, *i.e.*, words, are independent of each other given the context of the class – this is the so-called “Naïve assumption”. Because of this independence assumption, the parameters for each attribute can be learned separately, and this greatly simplifies learning, especially when the number of attributes is large (MCCALLUM & NIGAM, 1998).

Naïve-Bayes Classifiers are a family of algorithms based on the Bayes' theorem. Although their performances are usually not as good as some other statistical learning methods such as *nearest-neighbor classifiers* (YANG & CHUTE, 1994), *support vector machines* (JOACHIMS, 1998), and *boosting* (SCHAPIRE & SINGER, 2000), it is very efficient and easy to implement compared to other learning methods (MYAENG; HAN & RIM, 2006). A performance comparison of nearest-neighbors and naïve-bayes techniques can be found in (RASJID & SETIAWAN, 2017).

Bravey provides two algorithms are provided to recognize intent: (i) *fuzzy* – does not follow entity order, less precise but easier to hit with few training samples; (ii) *sequential* – process entities in strict sequential order, more precise but harder to hit with few training samples. We adopted the *fuzzy* algorithm because we want to recognize sentences with variations in the entity order. For instance, we consider the sentences “Given that I inform 10 to {Quantity}” and “Given that I fill {Quantity} with 10” equivalent.

Listing 17 presents an example in JavaScript that uses Bravey. Setup and use are simple. Firstly, we create an object that corresponds to the desired algorithm (fuzzy

or sequential). Then we can train the library by adding example sentences (called “documents”) and their correspondent intents. Whether we want to recognize specific entities – like a *number*, a *date*, a *time*, an *email* –, we can add the corresponding recognizer as an entity. For instance, we can add a time recognizer for the English language by adding an object from the class `Bravey.Language.EN.TimeEntityRecognizer`. Bravey currently supports English, Italian, and Portuguese, hence we can add recognizers for these languages. Custom recognizers can be defined when needed. Finally, to recognize entities in a sentence, we can call the method `test`, that receives a sentence and returns an object with information on the recognized intents, their position in the sentence, related scores (*i.e.*, probabilities), etc.

Listing 17 – Example in Bravey⁵⁴

```
// Using the Fuzzy NLP processor
var nlp = new Bravey.Nlp.Fuzzy();
var options = { fromFullSentence: true, expandIntent: true };
// Adding an example and the related intent for training
nlp.addDocument("I want a pizza!", "pizza", options );
// Testing if it can recognize the intent (pizza)
console.log( nlp.test("Want pizza, please").intent); // "pizza"
// Adding a new entity recognizer for numbers
nlp.addEntity(new Bravey.NumberEntityRecognizer("quantity"));
// Adding an example for training
nlp.addDocument("I want 2 pizzas!", "pizza", options );
// Testing
console.log( nlp.test(
    "Want 3 pizzas, please").entitiesIndex.quantity.value ); // 3
// Adding a time entity recognizer
nlp.addEntity(
    new Bravey.Language.EN.TimeEntityRecognizer("delivery_time"));
// Adding an example for training
nlp.addDocument( "I want 2 pizzas at 12!", "pizza", options );
// Testing
console.log( nlp.test(
    "Deliver 3 pizzas for 2pm, please").entitiesIndex );
// { delivery_time: { value: "14:00:00" }, quantity: { value:3 }, ... }
```

⁵⁴ Example adapted from the examples available at <https://github.com/BraveyJS/Bravey>

Bravey also allows to define entities and parameterize them in training documents using brackets. This facilitates the training since it is not needed to use all the different values, but only the respective entities. Listing 18 presents an example.

Listing 18 – Example of parameterization in Bravey⁵⁵

```
var nlp = new Bravey.Nlp.Fuzzy();
// Adding an intent "order_drink"
nlp.addIntent("order_drink", [
  { entity: "drink_name", id: "drink_type" },
  { entity: "number", id: "quantity" }
]);
// Creating an entity to represent a drink
var drinks = new Bravey.StringEntityRecognizer("drink_name");
drinks.addMatch("coke", "coke");
drinks.addMatch("coke", "cola");
drinks.addMatch("mojito", "mojito");
drinks.addMatch("mojito", "moito");
nlp.addEntity(drinks);
// Creating an entity to recognize a number
nlp.addEntity(new Bravey.NumberEntityRecognizer("number"));
// Adding training examples with entities
nlp.addDocument("I want {drink_name}!", "order_drink");
nlp.addDocument("I want {number} {drink_name}", "order_drink");
// Recognizing
console.log(nlp.test("Want a cola, please"));
// {intent:"order_drink", entities: [
//   {entity: "drink_type", value: "coke", ...} ] }
console.log(nlp.test("Want 2 mojitos, please"));
// {intent:"order_drink", entities: [
//   {entity: "drink_type", value: "mojito", ...},
//   {entity: "quantity", value: "2", ...} ] }
```

⁵⁵ Example adapted from the examples available at <https://github.com/BraveyJS/Bravey>

5.6.Intent recognition in Concordia

Concordia – the metalanguage introduced by this thesis – uses intent recognition for processing the following constructions:

- (i) *Variant sentences*, presented in the section 6.1.11.
- (ii) *Test Case sentences*, presented in the section 6.1.12.
- (iii) *Test Events' sentences*, presented in the section 6.1.13.
- (iv) *User Interface Element restrictions*, presented in the section 6.1.10.
- (v) *Database properties*, presented in the section 6.1.9.

Variant sentences and *Test Case sentences* must be expressed in the Given-When-Then format (see this format in the section 4.1.2) and use the first-person singular. We decided to adopt the first-person singular to reduce the number of variations when writing the sentences. The personal pronoun “I” represents the current *user* or *user role* – sentences are always declared from his/her/its point of view. We preferred to use first-person singular instead of, for example, third-person singular – as adopted by Soeken *et al.* (SOEKEN; WILLE & DRECHSLER, 2012) for generating class diagrams –, because sentences become more concise and less repetitive. A great part of these sentences describes interactions with a user interface.

User Interface Element (UIE) restrictions and *Database properties* usually contain names and values. For example, a Database may have a sentence such as “-username is "admin"” which defines the property “username” and the value “admin”. UIE restrictions may contain Otherwise sentences (see 6.1.10.1) to describe the expected behavior in case of the restriction is not satisfied. These sentences are recognized in the same way as Variant sentences.

Our approach uses dictionaries. Every spoken language can have its own dictionary – we currently use JSON files as storage media, *e.g.*, `pt.json` for Portuguese.⁵⁶ A dictionary defines *intents*, *entities*, and *training sentences*. Every entity contains words or small sentences that describe it. For example, the entity “click” can be described by the words “click”, “activate”, and “trigger” in a dictionary

⁵⁶ A dictionary can be customized by users whether needed, although we do not expect them to do so.

for the English language or by the words “clico”, “ativo”, “aciono” and “disparo” in a dictionary for the Portuguese language. A *training sentence* reinforces a certain intent by describing examples that contain words or entities (or a mix of words and entities). For example, the input sentence “Given that I activate {Save As}” would be recognized as the intent “click”, with an entity also called “click” and another entity identified (through a regular expression) as “Save As”. Thus, we can define different entities to form a certain intent and use training sentences to exemplify the intent for the learning algorithm.

Bravey removes diacritics and uses stemming (section 5.2.2) before comparisons. This greatly reduces the need for declaring different versions of the same word or training sentences.

In addition to the IR techniques, we created a validator for intents. Firstly, we define syntax rules for every intent. These rules constraint the type and number of entities accepted by a certain intent. For example, the sentence “Given that I inform {Name}, {Surname}, {Phone Number}, and {Address}”, has the intent “fill” and references to four UI Elements. Listing 19 shows the syntax rule for the intent “fill”, defined as a JSON object. The rule constraints the minimum and the maximum number of accepted entities (`minTargets` and `maxTargets`), the accepted entity types (`targets`), and details the number of accepted entities for every accepted type (`min` and `max`). Since the prior example has 4 UI Elements and the rules accepts up to 999, it passes the validation. However, a sentence like “Given that I inform {Name} with “Bob” and “Alice””, would not pass the validation, since it has two values and the rule defines a maximum of one.

Listing 19 – Example of a syntax rule for a intent

```
{ name: "fill",  
  minTargets: 0, maxTargets: 999,  
  targets: [ "ui_element", "ui_literal", "value", "number", "constant"  
],  
  ui_element: { min: 0, max: 999 },  
  ui_literal: { min: 0, max: 999 },  
  value: { min: 0, max: 1 },  
  number: { min: 0, max: 1 },  
  constant: { min: 0, max: 1 }  
},
```

5.7. Concluding remarks

This chapter summarized natural language processing techniques adopted by approaches related to ours, as well as common techniques and approaches from literature. It also presents a comparison of solutions for NLP and intent recognition and details the solution used by our approach, and how we used it.

6Concordia

If your requirements aren't changing it may be a sign that your stakeholders aren't interested in what you are building.
- Scott W. Ambler

This chapter presents Concordia, a novel metalanguage for agile requirements specification.

Concordia is the name of a Roman goddess who was the personification of “concord” or “agreement” (ENCYCLOPEDIA BRITTANICA, 2017). We decided to adopt that name to reinforce the idea of creating a specification in which stakeholders and the software team can discuss and agree.

Since Concordia is inspired in Gherkin (THE CUCUMBER TEAM, 2012) and keeps compatibility with that metalanguage, we expected that: (i) existing Gherkin feature files can be reused; (ii) Concordia may have a very low learning curve for Gherkin users; and (iii) Gherkin-based tools can be used to generate test skeletons for features that implement non-functional requirements.

Table 7 compares language constructions supported by Gherkin and Concordia. Section 6.1 details many of these language constructions and section 6.2 shows an example.

Table 7 - Language constructions in Gherkin and Concordia

Construction	Gherkin	Concordia	Max. declarations per file
<i>Comment</i>	✓	✓	not limited
<i>Language</i>	✓	✓	1
<i>Import</i>	×	✓	not limited
<i>Tag</i>	✓	✓	not limited
<i>Feature</i>	✓	✓	1
<i>State</i>	×	✓	not limited
<i>Scenario</i>	✓	✓	not limited
<i>Background</i>	✓	✓	not limited
<i>Constants</i>	×	✓	1
<i>UI Element</i>	×	✓	not limited
<i>Table</i> ⁵⁷	×	✓	not limited
<i>Database</i>	×	✓	not limited
<i>Variant</i>	×	✓	not limited
<i>Test Case</i>	×	✓	not limited
<i>Before All</i>	×	✓	1
<i>After All</i>	×	✓	1
<i>Before Feature</i>	×	✓	1
<i>After Feature</i>	×	✓	1
<i>Before Each Scenario</i>	×	✓	1
<i>After Each Scenario</i>	×	✓	1
Total	6	20	-

6.1. Language constructions

Figure 9 gives an overview of Concordia declarations. Rectangles with background in white (i.e., *Feature*, *Scenario*, *Background*, and *table DSL*) refer to declarations also available in Gherkin.

⁵⁷ Table is not only the DSL for parameterization presented in section 4.1.4, but a construction that also supports that DSL.

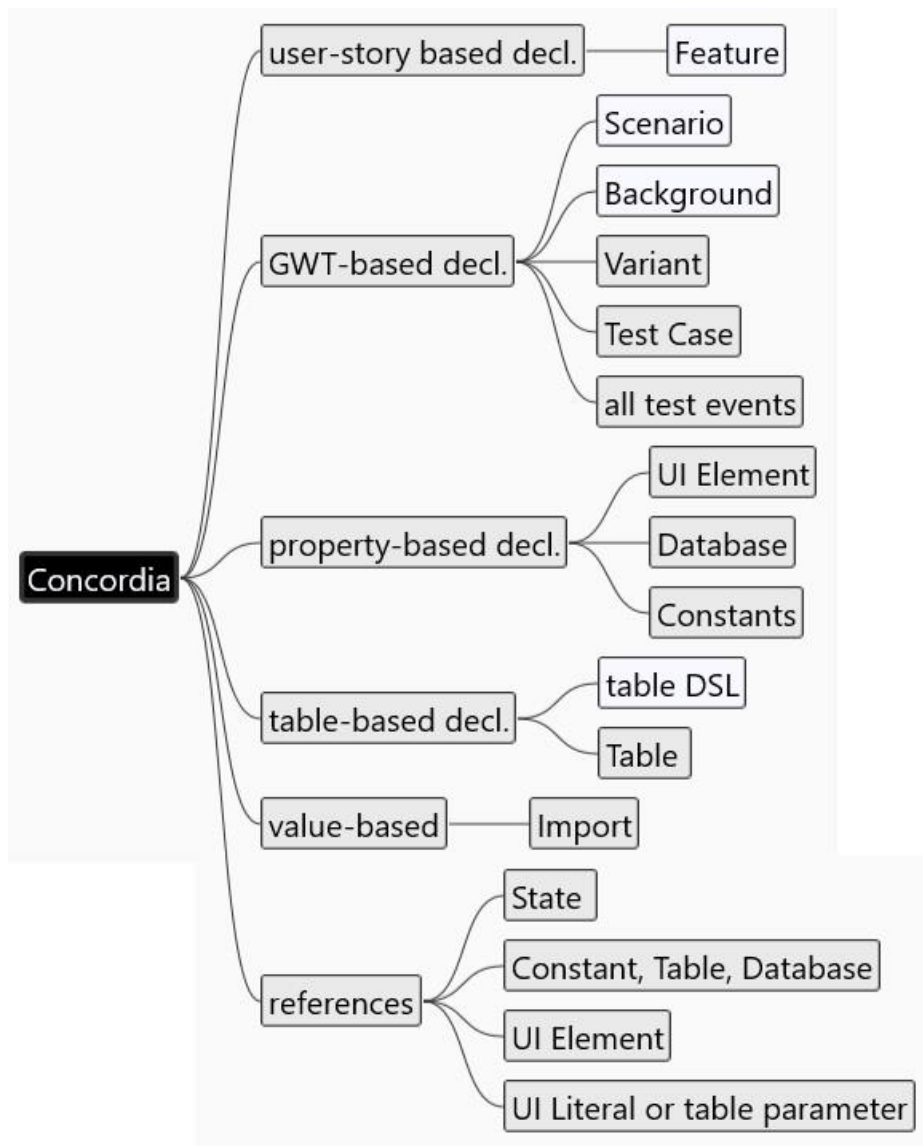


Figure 9 - Overview of Concordia declarations

Keywords and names in Concordia are case insensitive. Syntax is line-based and, therefore, keywords, names, values and other constructions must not be separated by line-breaks.

Table 8 presents symbols adopted in some language constructions. **Values** must be embraced with quotes, *e.g.*, "Hello". **Numbers** do not need quotes. References to **Constants**, **Tables** or **Databases** must be embraced with “[” and “]”, *e.g.*, [AppName]. **Script commands**, *i.e.*, database scripts or console scripts, (see 6.1.13) must be embraced with single quotes, *e.g.*, 'DELETE FROM sales'. **Queries** (section 6.1.10.2) may use quotes instead of single quotes, *e.g.*, "SELECT * FROM sales

WHERE date = '2017-12-25'', otherwise internal single quotes have to be escaped. **States** (see section 6.1.11) must be embraced with a tilde, *e.g.*, ~user is logged in~. **UI Literals**, *i.e.*, widget identifications, must be embraced by “<” and “>”, *e.g.*, <quantity>. References to **UI Elements** (section 6.1.10) must be embraced with brackets, *e.g.*, {Quantity}. **Tags** start with “@”, *e.g.*, @critical.

Table 8 – Symbols in Concordia

Example	Meaning
"hello"	Value
"SELECT * FROM sale"	Query (a value that starts with select)
'DELETE FROM sale'	Script Command
[AppName]	Reference to a Constant, Table or Database
<quantity>	UI Literal
{Quantity}	Reference to a UI Element
~user is logged in~	State
@critical	Tag

Available data types are String, Integer, Double, Boolean, Date, Time, and DateTime. They are inferred from declared Constants (section 6.1.7), User Interface Element properties' values (section 6.1.10), and Tables (section 6.1.8). Date values must adopt the format “YYYY-MM-DD” or “YYYY/MM/DD” where “YYYY” represents a four-digit year, “MM” represents a two-digit month, and “DD” represents a two-digit day. Time values must adopt the format “HH:NN:SS” or “HH:NN”, where “HH” represents a two-digit hour, “NN” represents a two-digit minute, and “SS” represents a two-digit second. DateTime values must also adopt these formats for their date part and time part, respectively. Internationalization, *i.e.*, to use formats according to a country or region, may be supported in future versions of the metalanguage.

6.1.1.Comment

A comment makes a content to be ignored (for processing purposes). Concordia supports line comments, like those in Listing 20, which makes the content at the right of the used symbol to be ignored. A line comment in Concordia starts with a hashtag (#).

Listing 20 – Comment in Concordia

```
# This is a comment
```

```
Feature: Pay with credit card # This is also a comment
```

6.1.2.Language

The language used in the current specification file can be defined by a special comment, which starts with the keyword “language” and is followed by a colon and an ISO 639-1 language code. Listing 21 shows an example with the language construction for Portuguese (pt). English (en) is assumed whether the language is not defined in the document nor is parameterized to the parser.

Listing 21 – Language in Concordia

```
#language: pt
```

```
Funcionalidade: Pagar com cartão de crédito
```

6.1.3.Import

An import allows using declarations from feature files. A feature file can be imported by its full path or relative path – like the examples in Listing 22.

Listing 22 – Import in Concordia

```
import "/path/to/buy-product.feature"
```

```
import "../..find-product.feature"
```

6.1.4.Tag

A tag adds information to a language construction. It can be used in Features, Scenarios, UI Elements, Variants, and Test Cases, for referencing other constructions, representing cross-cutting concerns, or defining filterable content. Tags start with “@” and can receive numeric or textual parameters, embraced with parenthesis and separated by comma. Listing 23 shows some examples.

Listing 23 – Tags in Concordia

```
@slow @report
```

```

Feature: Generate sales report

@importance(8) @issue(2, Alice)
Scenario: Report sales by month

```

Table 9 presents reserved tags, *i.e.*, tags with defined purpose and syntax.

Table 9 - Reserved tags

Reserved tag	Purpose	Example
<i>extends</i>	Allows a UI Element to inherit the properties of another UI Element.	@extends(Full name)
<i>fail</i>	Indicates that a Test Case should fail.	@fail
<i>generated</i>	Indicates that a Test Case was generated by a computer.	@generated
<i>global</i>	Indicates that a UI Element is global.	@global
<i>ignore</i>	Allows ignoring a Feature, Scenario, Variant or Test Case.	@ignore
<i>importance</i>	Defines the importance of a Feature, Scenario, Variant or Test Case, varying from 1 (lowest) to 9 (highest).	@importance(8)
<i>scenario</i>	Allows a Test Case to reference a Scenario by its index.	@scenario(1)
<i>variant</i>	Allows a Test Case to reference a Variant by its index.	@variant(2)

6.1.5.Feature

A feature is represented like in Listing 24. It may have one or more scenarios. The user story part is optional.

Listing 24 – Feature in Concordia

```

Feature: Generate sales report
    As a sales manager
    I would like to generate a sales report
    In order to keep abreast of company sales

```

Feature names are *global*, *i.e.*, a specification cannot have a repeated feature name.

6.1.6.Scenario

A scenario is represented like in Listing 25. Steps are optional, must adopt the Given-When-Then format, and should be described from a high-level, business point of view. A scenario may have one or more Variants (section 6.1.11). Scenario names must be unique inside a Feature.

Listing 25 – Scenario in Concordia

```
Feature: Print sales report
Scenario: Print directly
    Given that I have generated the sales report
    When I trigger the option to print
    Then I see a message that the report was sent to the default printer
```

6.1.7.Constants

A Constants block allows to define constant values that can be used in Variant steps, Test Case steps, User Interface Element properties, and queries. Every constant declaration starts with a dash (-). Constants' names are embraced with quotes. Data types of constant values are inferred. Listing 26 shows an example with a Constants block.

Listing 26 – Constants in Concordia

```
Constants:
    - "PI" is 3.14159
    - "AppName" is "My App"
```

Constants' names are *global*, *i.e.*, a specification cannot have a repeated constant name, and they share the same namespace with Tables (section 6.1.8) and Databases (6.1.9). A Constant can be referenced by its name inside “[” and “]”, *e.g.*, [AppName].

6.1.8.Table

A Table defines values that can be used in properties of User Interface Elements for creating dynamic constraints or generating dynamic test data. It works like a consultable in-memory table. Listing 27 shows an example of a Table.

Listing 27 – Table in Concordia

Table: Users			
	login	password	
	bob	bob123456	
	alice	411c3p4s\$	

The first row of a Table defines column names, and the other rows define the corresponding values. Data types are inferred according to the given values.

Tables names are *global*, *i.e.*, a specification cannot have repeated table names, and share the same namespace with Constants (section 6.1.7) and Databases (6.1.9). A Table can be referenced by its name inside “[” and “]”, *e.g.*, [Users].

6.1.9.Database

A Database represents an external data source, *i.e.*, a database or a file. Listing 28 shows two examples: one with a MySQL database and the other with JSON file.

Listing 28 – Database in Concordia

Database: TestDB	
-	type is "mysql"
-	name is "testdb"
-	host is "localhost"
-	username is "tester"
-	password is "test123"
Database: TestDB2	
-	type is "json"
-	path is "/path/to/testdb.json"

Properties start with a dash (-) and their values must be embraced with quotes. Table 10 presents the database properties.⁵⁸

Table 10 - Database properties

Property	Description	Required	Default value
<i>type</i>	Database type	Yes	No
<i>host</i>	URL, DSN, or IP of the database	No	“localhost”
<i>port</i>	Database port	No	Vary
<i>name</i>	Database name	Vary	No
<i>path</i>	Database path or file path	Vary	No
<i>username</i>	Username	No	Vary
<i>password</i>	Password	No	Vary
<i>options</i>	Database options	No	Vary

Database names are *global*, *i.e.*, a specification cannot have repeated database names, and share the same namespace with Constants (section 6.1.7) and Tables (section 6.1.8). A Database can be referenced by its name inside “[” and “]”, *e.g.*, [TestDB].

6.1.10. UI Element

A UI Element represents a widget that belongs to a Feature. It can define related constraints and business rules through properties. Table 11 presents available properties.⁵⁹

⁵⁸ Table cells with “Vary” means “vary according to the database type”. For instance, default value for “port” can be 3050 whether the property “type” is “mysql”, or “5432” whether “type” is “postgres”.

⁵⁹ More details in <https://github.com/thiagodp/concordialang/blob/master/docs/language/en.md#user-interface-element>

Table 11 - UI Element properties

Property	Description	Required	Default value	Otherwise steps ⁶⁰
<i>id</i>	Widget identification.	No	The UI Element name in lowercase ⁶¹	No
<i>type</i>	Widget type.	No	textbox	No
<i>editable</i>	Whether it can accept input data.	No	Auto-detected ⁶²	No
<i>data type</i>	Data type (see 6.1).	No	string	No
<i>value</i>	Value, list of possible values, query to retrieve possible values, or other UI Element that produces the value.	No	Auto-generated.	Yes
<i>minimum length</i>	Minimum length, query to retrieve it, or other UI Element that produces it.	No	No	Yes
<i>maximum length</i>	Minimum length, query to retrieve it, or other UI Element that produces it.	No	No	Yes
<i>minimum value</i>	Minimum value, query to retrieve it, or other UI Element that produces it.	No	No	Yes
<i>maximum value</i>	Minimum value, query to retrieve it, or other UI Element that produces it.	No	No	Yes
<i>format</i>	Regular expression that defines the format.	No	No	Yes
<i>required</i>	Whether is required to inform a value.	No	False	Yes

⁶⁰ Indicates whether the property can have Otherwise steps defined, in order to state what is supposed to happen when an input data violates the property.

⁶¹ The default can be changed from Camel Case (e.g., “fullName”) to keep the original name as is or to change it to Pascal Case (e.g., “FullName”), Snake Case (e.g., “full_name”), or Kebab Case (e.g., “full-name”).

⁶² Editable will be automatically true when the property *type* is “checkbox”, “fileInput”, “select”, “table”, “textbox”, or “textarea”.

Listing 29 presents a simple UI Element named “Full Name”. By default, it assumes the type “textbox”, the data type “string”, and the identification “fullName” (*i.e.*, the name in camel-case).

Listing 29 – A simple UI Element in Concordia

UI Element: Full Name

UI Element names must be unique inside a Feature (section 6.1.5). A *global* UI Element can be defined by adding the tag “@global” (see section 6.1.4). In this case, there must not exist two global UI Elements with the same name. A UI Element from a Feature has precedence over a global UI Element.

Inheritance is possible through the tag “@extends” (see section 6.1.4), *e.g.*, @extends(Full Name). The UI Element with that tag receives the properties from the referenced UI Element.

6.1.10.1. Otherwise steps

Column “Otherwise Steps” from Table 11 indicates whether the property can have Otherwise steps defined. *Otherwise steps* state what is supposed to happen when an input data violates the property. These steps have the same syntax than *Then* steps (from GWT format) and do not start with a dash. Listing 30 illustrates the definition of two UI Elements that contain properties with Otherwise steps.

Listing 30 – UI Element with Otherwise steps

UI Element: Profession <ul style="list-style-type: none"> - type is select - value is in ["Professor", "Engineer", "Accountant"] Otherwise I must see "The given profession is not allowed." - required is true Otherwise I must see "Please inform the profession." UI Element: Annual Salary <ul style="list-style-type: none"> - data type is double - minimum value is 12000.00

Otherwise I must see "Salary must be greater than or equal to 12000.00"

6.1.10.2. Dynamic values

A UI Element property can have a dynamic value, to support cases in which the value is only known at run-time. The values can be retrieved from a Table (section 6.1.8), a Database (section 6.1.9) or another UI Element. Instead of defining another DSL to query them, we decided to adopt Structured Query Language (SQL) because of its broad use in computer science and even business (*i.e.*, business managers may need to know SQL to create customized reports in report generators).

In Concordia, queries can have references to Tables (section 6.1.8), Databases (section 6.1.9), Constants (section 6.1.7) or User Interface Elements (section 6.1.10). These references are checked when a query is processed and then transformed to their corresponding values.

Listing 31 illustrates a case in which the properties “minimum value” and “maximum value” of the UI Element “Salary” vary according to the value of the UI Element “Profession”. The value of “Profession” is also retrieved dynamically, through a query.

Listing 31 – UI Element with dynamic properties

```

UI Element: Profession
- type is select
- value comes from "SELECT name FROM [Professions]"
  Otherwise I must see "The given profession is not allowed."
- required is true
  Otherwise I must see "Please inform the profession."

UI Element: Annual Salary
- data type is double
- minimum value comes from "SELECT min_salary FROM [Professions]
  WHERE name = {Profession}"
  Otherwise I must see "Salary is lower than the minimum value."
- maximum value comes from "SELECT max_salary FROM [Professions]
  WHERE name = {Profession}"
  
```

Otherwise I must see "Salary exceeded the maximum value."

Table: Professions

name	min_salary	max_salary
Professor	12000.00	240000.00
Engineer	15000.00	350000.00
Accountant	13000.00	260000.00

In the above example, the properties query their values from a Table (section 6.1.8) named “Professions”. Instead, they could query them from an external data source, defined as a Database (section 6.1.9). Queries of “Annual Salary” have references to the UI Element “Profession”. These references make the value generator to produce a value for “Profession” prior to running the query.

6.1.10.3. References

A reference to a UI Element is denoted by a name between brackets, *e.g.*, {Profession}. References to UI Elements from other features must contain the feature name and the UI Element name separated by two colons, *e.g.*, {Register Employee::Profession}. References can be used in queries, Variant steps (section 6.1.11), and UI Element properties.

6.1.11. Variant

A Variant allows expressing interactions between a user (or user role) and the system, in order to perform a Scenario. It also serves as a template for generating Test Cases – it always generates at least one Test Case. The name “Variant” was inspired in Tartare (TELEFÓNICA, 2016), a testing framework that adopts data-driven tests for scenarios and names every input variation as a variant. In Concordia, Variants represent variations of a same Scenario.

Variant steps must be expressed in the Given-When-Then format and use the first-person singular. We decided to adopt the first-person singular for Variants and Test Cases to reduce the number of variations when writing the steps. The personal

pronoun “I” represents the current *user* or *user role*. Steps always declare expectations from his/her/its point of view. Listing 32 shows an example of a Variant. Variant steps may contain values, numbers, constants, states, script commands, UI Literals, and references to UI Elements

Listing 32 – Variant in Concordia

```
Variant: Usual login
  Given that I am on the [Login Page]
  When I inform my {Username} and {Password}
    and I click on {OK}
  Then I see a [Welcome Message]
    and I have ~user logged in~
```

The step in which a State is declared changes its meaning:

- *Given* step: the state is *required*;
- *When* step: the state is *called*;
- *Then* step: the state is *produced* (like in Listing 32).

When a state is declared in a Given step or in a When step, it must exist in a Then step of another Variant. Whether that Variant belongs to another feature, that feature must be imported. Otherwise, the compiler will not be able to locate the referenced state.

6.1.12. Test Case

A Test Case is a kind of low-level Variant that contains generated combinations of test scenarios, test data, and test oracles. It represents a test case that interacts with the system through its UI, belongs to a Feature, and may have references to a Scenario and a Variant through tags. Test Cases can be declared in `.feature` files, but since they have a lower abstraction level than Scenarios and Variants, we encourage the declaration in `.testcase` files.

When a Test Case is generated from a Variant:

- Constants are replaced with their declared values – for instance, `[PI]` is replaced with `3.1416`;

- References to UI Elements are replaced with UI Literals – for instance, {Full Name} is replaced with <#fullName>;
- Steps that have actions able to enter input data (*e.g.*, “fill”, “select”, “append”, “attach file”) but do not declare the data (*i.e.*, value, number or constant), are going to receive data. Whether the action target is a UI Literal, it receives a random test data – for instance, the step “When I fill <#fullName>” may become “When I fill <#fullName> with “%8A#~kT^1””. Whether the action target is a UI Element, it receives a value that varies according to the approach’s target data test case – for instance, the step “When I fill {Full Name}” may become “When I fill <#fullName> with “”” whether the target data test case is generating an empty value.
- States in Given steps and When steps are replaced by their producers’ steps, that is, by the steps from the Test Cases that produce the required states;
- Then-steps with state are removed.

Listing 33 shows an example of a Test Case.

Listing 33 – Test Case in Concordia

```
Test Case: Usual login - 1
Given that I am on "http://localhost/myapp"
When I fill <#username> with "alice"
    and I inform <#password> with "4l1c3p4s$"
    and I click on <#ok>
Then I see "Welcome"
```

6.1.13. Test Events

Concordia supports test events for environment configuration (*e.g.*, preparing or adjusting the application’s database, running commands on console). Table 12 presents them and Listing 34 shows an example.

Table 12 - Test Events

Event	When it occurs
-------	----------------

<i>Before All</i>	Before all the tests start.
<i>After All</i>	After all the tests execute.
<i>Before Feature</i>	Before running the tests of a certain feature.
<i>After Feature</i>	After running the tests of a certain feature.
<i>Before Each Scenario</i>	Before running a test of a scenario.
<i>After Each Scenario</i>	After running a test of a scenario.

Listing 34 – Test Events in Concordia

```

Feature: Login
# ...
Before Each Scenario:
    When I run the script 'DELETE FROM [TestDB].user'
        And I run the script 'INSERT INTO [TestDB].user (username, password) VALUES ("bob", "bob123"), ("alice", "4l1c3p4s$")'
```

6.2.A quick example

Listing 35 presents an example that gives an overview of the metalanguage. The example specifies the login for a web application. The Feature and the Scenario are written from a high-level perspective, while the Variant and the other declarations give more details on the expected behavior.

Listing 35 – A Quick Example in Concordia

```

Feature: Login
    As a user
        I would like to authenticate myself
        In order to access the application

Scenario: Successful login
    Given that I can see the login screen
    When I enter with valid credentials
    Then I can access the application's main screen

Variant: Usual login
    Given that I am in the [Login Screen]
```


<p>When I fill {Username}</p> <p>And I fill {Password}</p> <p>And I click on {OK}</p> <p>Then I see "Welcome"</p> <p>And I have ~user is logged in~</p> <p>Constants:</p> <ul style="list-style-type: none"> - "Login Screen" is "http://localhost/app/login" <p>Table: Users</p> <table> <tr> <td> username password </td> </tr> <tr> <td> bob 123456 </td> </tr> <tr> <td> alice 411c3pass </td> </tr> </table> <p>UI Element: Username</p> <ul style="list-style-type: none"> - value comes from "SELECT username FROM [Users]" Otherwise I must see "Invalid username or password." - required is true Otherwise I must see "Please inform the username." <p>UI Element: Password</p> <ul style="list-style-type: none"> - value comes from "SELECT password FROM [Users] WHERE username = {Username}" Otherwise I must see "Invalid username or password." - required is true Otherwise I must see "Please inform the password." <p>UI Element: OK</p> <ul style="list-style-type: none"> - type is button 	username password	bob 123456	alice 411c3pass
username password			
bob 123456			
alice 411c3pass			

6.3.Concluding remarks

This chapter provided an overview of Concordia by describing its syntax through examples. It also presented Concordia's basic concepts and compared Concordia and Gherkin with respect to their supported language constructions.

7 Approach

Stay committed to your decisions, but stay flexible in your approach.

- Tony Robbins (American writer)

This chapter details the proposed approach to mitigate the problems and gaps identified in previous chapters.

7.1.Overview

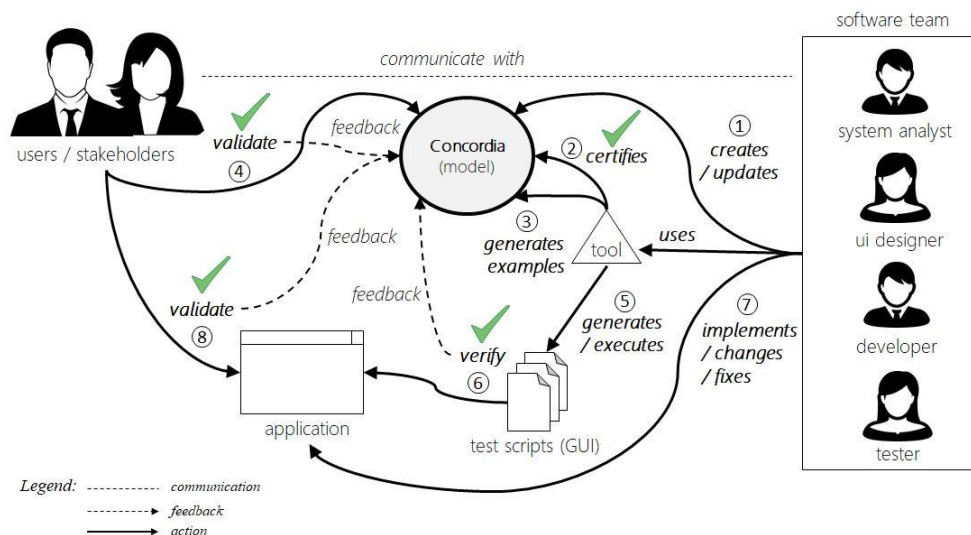


Figure 10 - Overview of the process

Figure 10 illustrates the process involved with the proposed solution. Software teams and stakeholders collaborate to create a shared understanding of needs, desires, concerns, and related solutions. Requirements specification documents serve as communication media, to capture that shared understanding in the form of features and scenarios. Functional and non-functional requirements specifications guide development and testing activities. Feedback is a fundamental source of knowledge for validating and improving requirements and the application. Collaborative work avoids communication problems and contributes to ensuring that the

solution considers different points of view and that all the participants understand the requirements specifications.

In this context, Concordia is used as a central model that enables the proposed process, and the tool supports the involved activities for verification and validation. After a requirements elicitation session with stakeholders, software team members can meet to specify collaboratively (step 1) the features needed for the next release (the desired increment for the next version). The team then reviews the produced Concordia specification informally and uses the tool for checking errors (step 2) and producing examples in the form of test cases (step 3). These test cases can give stakeholders examples on how the system should behave in specific scenarios, such as error handling scenarios. The verified specification is now ready for discussion with stakeholders. The team uses their feedback to validate the specification, before starting any development activities (step 4). Whether the stakeholders are not available for validation, the team should evaluate the risk of specified features and scenarios for the business, also considering their body of knowledge and experience. Whenever possible, the team should avoid taking the risk of developing features without prior feedback about their requirements. After validation (or taking the risk), the team can use the tool to generate functional test scripts for the application (step 5). The tool executes the test scripts and reports any nonconformance between the application and its specifications (step 6). If the team did not implement the features and scenarios in question, it can use the test scripts to drive their development. In this case, the team creates the scenarios incrementally to pass the tests. Anyway, the test scripts give the team feedback about any changes in the application (step 7), in the form of new or regression tests. When the application passes all the tests and it is ready for being released, the team schedules a validation meeting with stakeholders or sends them the application for validation. Finally, whatever the feedback received (step 8), the team reflects it in the specification. For example, whether stakeholders found a bug, the team can specify a Concordia test case aiming to confirm the sequence of inputs that caused the bug, before any changes the application. A change in the specification restarts the process (step 1).

7.2.High-level architecture

Figure 11 illustrates the architecture of the proposed solution. It defines interactions among the software team, stakeholders, documents, tools, and the system under test (SUT). Documents are referenced by letters from A to I. Generated documents are represented with icons in gray. The tool was divided into pieces that represent its roles, numbered from 1 to 5. The *verifier* (1) is responsible for checking the specification for problems (as those detailed in section 7.4.2). It uses training sentences and a dictionary for recognizing sentences of Concordia specifications, considering the target spoken language. The *test case generator* (2) uses these specifications to produce functional test cases in natural language. These test cases can serve as examples for validation with stakeholders and as models for producing test scripts. The software team can write additional test cases using Concordia if they need. Test cases can use external test data from test databases to simulate real (production-like) data. The number of generated test cases can be reduced by minimization and prioritization strategies (section 7.3.2), aiming to make the test time feasible. The *abstract test script generator* (3) converts all the Concordia test cases into abstract test scripts. A plug-in that works as a *test script generator and executor* (4) transforms the abstract test scripts into source code for a particular testing framework. These test scripts (source code) executes the SUT according to the test cases – simulating user inputs – and produces a report. The plugin converts the report (file) into the format expected by the tool. The *results analyzer* (5) compares these results with the expectations and produces a user-readable report (on the screen or onto a file). Eventually, other tools may consume that report.

In our vision, a tool implemented according to this approach and architecture must be easy to install and use – *i.e.*, must have very few setup steps, commands to run, and parameters to remember – and may adopt convention over configuration whenever possible, *e.g.*, default directories, algorithms, and patterns. These characteristics aim to facilitate the tool’s adoption, mainly for companies without prior experience with test automation. Members of the software team should be able to adopt Concordia and start using its automated tests without knowing much about how it works.

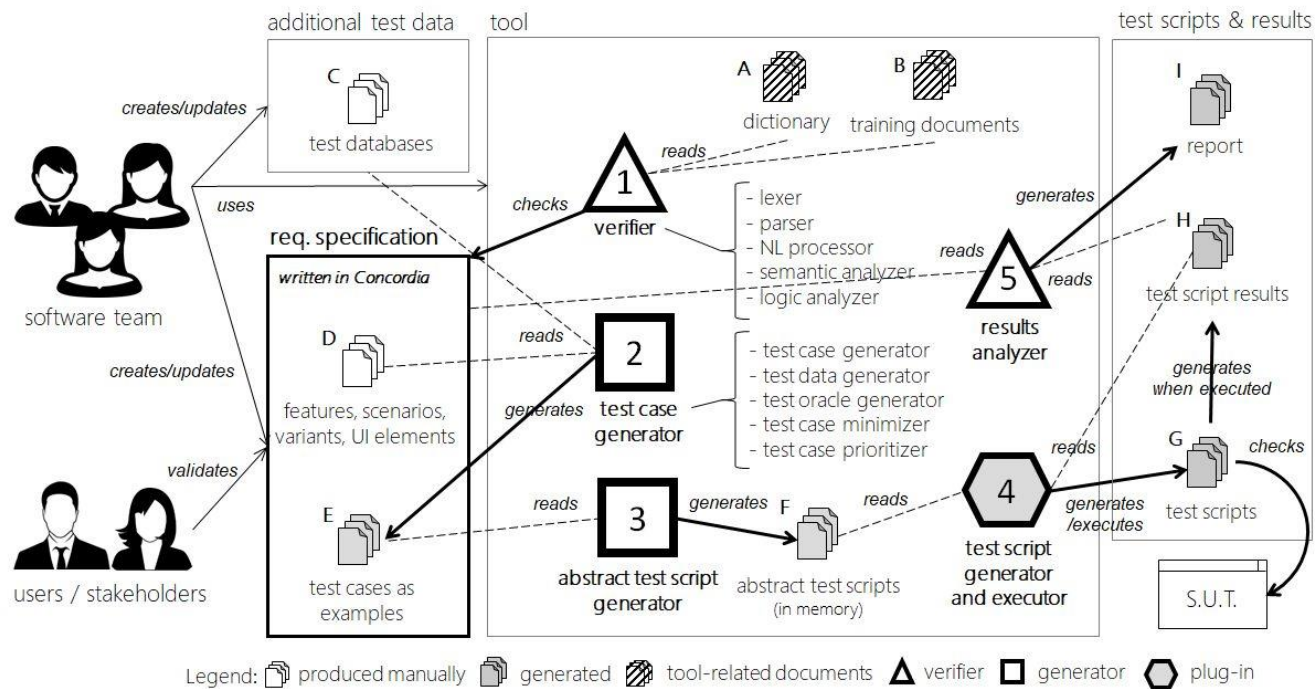


Figure 11 – High-level architecture

7.3.Verification

This section details problems concerning the verification through test cases and describes how the proposed approach deals with them.

7.3.1.Combinatorial explosion

The combinatorial explosion problem is associated with the difficulty to execute or to verify all the paths of an application. Non-trivial applications often have a very large number of possible paths and there is an overhead to execute them, principally in feasible time (ANAND et al., 2013). Whereas that reducing the paths to verify can make the execution time feasible, there are coverage losses in paths that could expose defects. Thus, it is important to have reduction criteria that adequately balance time and coverage.

Over the years, many prioritization techniques proposed to select a small set of test cases that can offer the highest path coverage (ELBAUM; MALISHEVSKY & ROTHERMEL, 2002; ROTHERMEL et al., 1999; SRIKANTH; WILLIAMS & OSBORNE, 2005). Reducing the number of test cases without sacrificing coverage may make test execution viable.

Notwithstanding, *input data sets* also suffer from the same problem. Besides the classical approaches – such as equivalence partitioning classes, limit-value analysis, and random generation (BEIZER, 2003; MYERS; THOMAS & SANDLER, 2011) –, many approaches based on *combinatorial methods* were proposed, with good results. Considering the classification from Cohen *et al.* (2007), there are *search-based approaches* (AHMED & ZAMLI, 2011; CHEN et al., 2009b, 2010; COLBOURN et al., 2010; NURMELA, 2004; SHIBA; TSUCHIYA & KIKUNO, 2004), *algebraic approaches* (CALVAGNA & GARGANTINI, 2008; HARTMAN, 2005; WILLIAMS, 2002; YAN & ZHANG, 2008), those based on *greedy algorithms* (COHEN et al., 1997; CZERWONKA, 2006; LEI & TAI, 1998; TUNG & ALDIWAN, 2000), and those that *mix* algebraic and greedy approaches (LEI et al., 2008; SHERWOOD, 1994).

In the following sections, we present the techniques adopted for mitigating the combinatory explosion.

7.3.2. Selection, minimization, and prioritization

A systematic mapping study by Catal and Mishra (2013) points out techniques for improving the cost-effectiveness of the testing activity, especially those related to regression testing:

- A. *Test Suite Minimization / Test Suite Reduction*: remove redundant test cases permanently to reducing the size of the test suite;
- B. *Test Case Selection / Regression Test Selection*: select some of the test cases and focus on the ones that changed parts of the software. They do not remove test cases, but selects test cases that are related to the changed portion of an artifact (*e.g.*, related to a change in the source code);
- C. *Test Case Prioritization*: identifies the efficient ordering of test cases for maximizing specific properties, such as the *failure detection rate* or *coverage rate*.

Techniques in A and B reduce testing time, but they can omit test cases that may detect certain types of defects (DO et al., 2010). Techniques in C, on the other side, do not omit test cases – which may make them unfeasible for large systems – and reduce test time by using parallelization of testing activities (DO et al., 2010). Yoo and Harman (2012) discuss all these techniques in deep. Since our approach involves test case *generation* and test case *execution*, different techniques are used by these operations.

Most prioritization techniques currently available in the literature are primarily focused on improving regression testing efforts using white box, code level and coverage-based approaches (CATAL & MISHRA, 2013; SRIKANTH; HETTIARACHCHI & DO, 2016). Since a software system is built upon its requirements, to utilize requirements information can potentially help identifying more important or error-prone test cases than just using source code information (SRIKANTH; HETTIARACHCHI & DO, 2016). Our work proposes a prioritization strategy based on the *importance of requirements for business stakeholders*.

Figure 12 condenses techniques for prioritization, according to Mohanty *et al.* (2011). The ones based on requirements and their risk are in accordance with the study by Srikanth *et al.* (2016), which identifies six prioritization criteria: *customer priority* (CP); *implementation complexity* (IC); *failure proneness* (FP); *requirements volatility* (RV); *business-criticality* (BC); and *random* (RD). The study concludes that:

- i) RD is the most used technique in the industry;
- ii) All the criteria were, individually or combined, better than RD;
- iii) Individually, FP had the best results, followed by CP;
- iv) The best combination was FP and CP;
- v) There is a strong correlation between FP and CP.

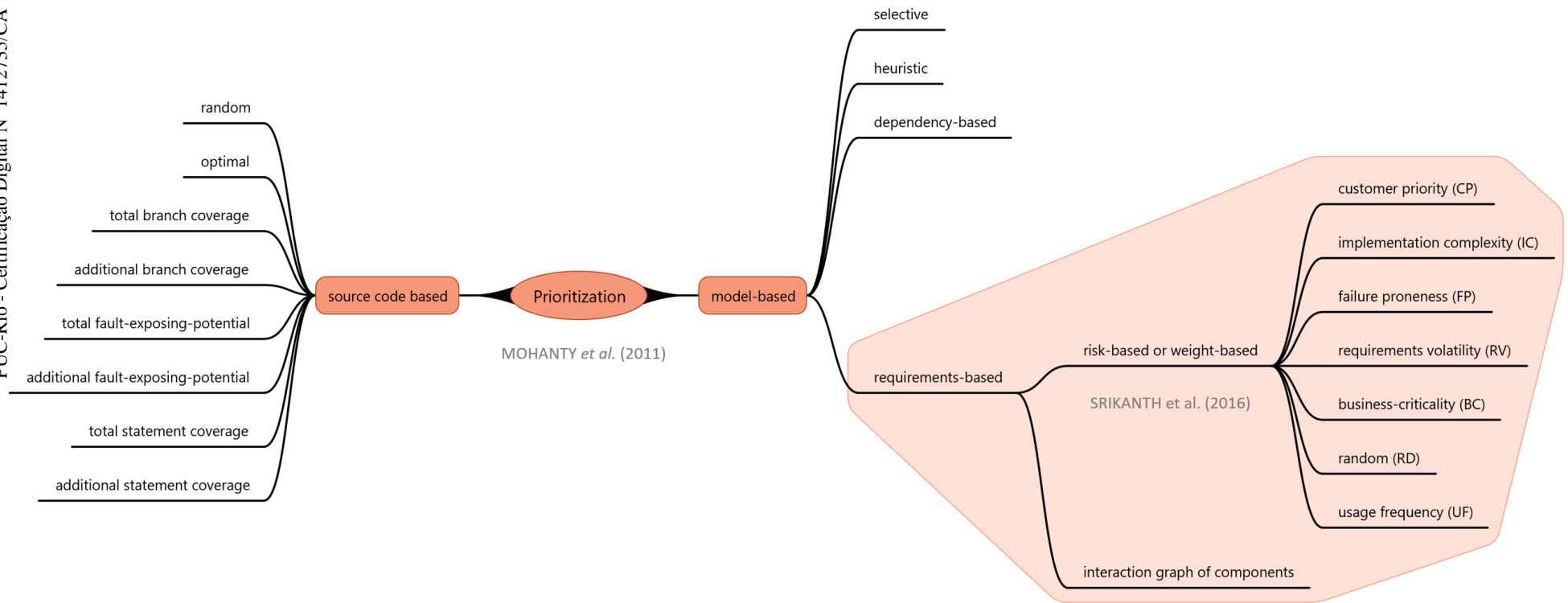


Figure 12 - Prioritization techniques

In a prior work (PINTO & STAA, 2013), we adopted a prioritization criterion based on an *importance* value, computed from Business Criticality, Implementation Complexity, and a *Usage Frequency* (UF) – not considered by Srikanth and colleagues (2016). Although we could not evaluate its effectiveness in practice, we believe that UF has correlation with the *failure detection rate*. The reason is simple: frequently used artifacts become well tested over time.

Despite the aforementioned prioritization criteria are promising and their combination potentially effective, we preferred to adopt a single *importance value* that can be attributed freely by business stakeholders or the development team to some constructions of the requirements specification. It is up to them to decide on the criterion that best fits their needs. Our recommendation, however, is *to use the customer priority by default*, since it is one of the best strategies available (SRIKANTH; HETTIARACHCHI & DO, 2016) and encourages customer involvement. The reasons to use a single, adjustable criterion are twofold:

- a) *Simplicity*: textual specifications should be simple to understand and write. Adopting more than one criterion may confuse readers and make prioritization more complex, and thus difficult to use;
- b) *Flexibility*: different projects may have different needs. Customer Priority may apply better to some projects, while Usage Frequency, Failure Proneness or Business Criticality may fit better to others, for example. Srikanth *et al.* (2016) found that for applications that are being released for the first time (*i.e.*, version 1.0), in which the software team lacks field data, CP can fit better than FP, while FP had better results for released software (post version 1.0). Making users aware of the candidate criteria and letting them choose which to use (for the importance value) may help them to achieve superior results, compared to adopt a single criterion for all their projects.

Therefore, our approach uses *importance* values for:

- i) Classifying features, scenarios, and variants (see 6.1.11);
- ii) Reducing the amount of generated test cases, *i.e.*, it minimizes the test suite;
- iii) Selecting (prioritizing) the test cases to be executed.

The evaluation of the impact of importance values in produced or executed tests is out of the scope of this thesis.

7.3.3. Combination strategies

Different studies (BELL, 2006; KUHN & REILLY, 2003; KUHN; WALLACE & GALLO, 2004; WALLACE D R, 2001) found that few combinations of inputs are needed to detect defects in production. Figure 13 illustrates this finding. Inspired by it, different approaches for combinatorial testing were proposed –e.g., those in the surveys from Grindal *et al.* (2005) or Nie & Leung (2011). Most of them use algebraic or greedy algorithms to produce *pairwise* (2-wise) or *t-wise* (a.k.a. n-wise) combination of elements.

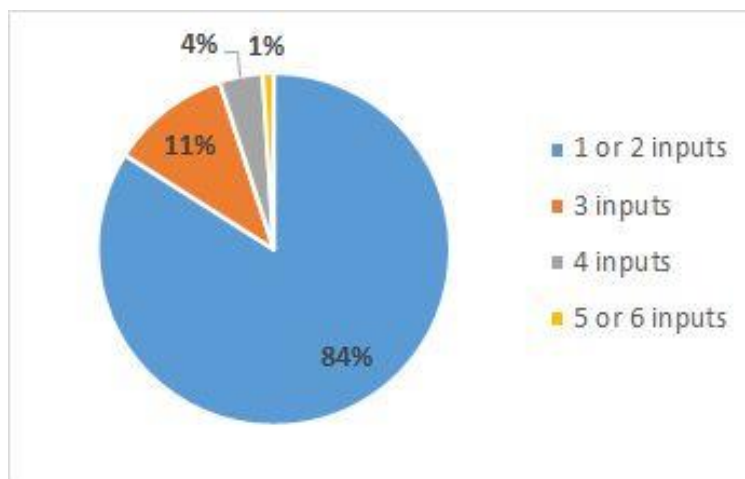


Figure 13 - Number of inputs needed to detect defects

Figure 14 exemplifies how T-wise combinations work. Briefly, the approach combines elements one-by-one, or two-by-two, or three-by-three, and so on, depending on the value of T , e.g., whether T is 2 (2-wise), the combinations are two-by-two.

t-wise

Example:

parameters and values

A	<i>X, Y</i>
B	<i>W, Z</i>
C	<i>I, J, K</i>

$2 \times 2 \times 3 = 12$ tests

100% coverage

Every *single* value
appears at least once

1-wise combination

#	A	B	C
1	<i>X</i>	<i>W</i>	<i>I</i>
2	<i>Y</i>	<i>Z</i>	<i>J</i>
3	<i>X</i>	<i>W</i>	<i>K</i>

3 testes

25% coverage

Every *pair* of values
appears at least once

2-wise combination

#	A	B	C	(AC)
1	<i>X</i>	<i>W</i>	<i>I</i>	X + I
2	<i>X</i>	<i>W</i>	<i>K</i>	X + K
3	<i>X</i>	<i>Z</i>	<i>J</i>	X + J
4	<i>Y</i>	<i>Z</i>	<i>I</i>	Y + I
5	<i>Y</i>	<i>Z</i>	<i>J</i>	Y + J
6	<i>Y</i>	<i>Z</i>	<i>K</i>	Y + K
7	<i>Y</i>	<i>W</i>	<i>J</i>	

7 tests

~58% coverage

Every *triplet* of values
appears at least once

3-wise combination

#	A	B	C
1	<i>X</i>	<i>W</i>	<i>I</i>
2	<i>X</i>	<i>W</i>	<i>J</i>
3	<i>X</i>	<i>W</i>	<i>K</i>
4	<i>X</i>	<i>Z</i>	<i>I</i>
5	<i>X</i>	<i>Z</i>	<i>J</i>
6	<i>X</i>	<i>Z</i>	<i>K</i>
7	<i>Y</i>	<i>W</i>	<i>I</i>
8	<i>Y</i>	<i>W</i>	<i>J</i>
9	<i>Y</i>	<i>W</i>	<i>K</i>
10	<i>Y</i>	<i>Z</i>	<i>I</i>
11	<i>Y</i>	<i>Z</i>	<i>J</i>
12	<i>Y</i>	<i>Z</i>	<i>K</i>

100% coverage

Note: Since there are 3 parameters, 3-wise covers all the combinations.

Figure 14 – Example of a T-wise combination

The chart in the Figure 13 shows that to use combinations of 1 or 2 inputs can make tests to detect up to 84% of applications' defects. For combinations of 3 inputs, 95%. *However, the smaller the number of combinations, the faster is the time to run the tests.* For that reason, we defined a set of **combination strategies** to use in different stages of our approach. Given two set of elements, A and B :

1. *One-wise*: Performs a one-wise combination of the elements from A and B ;
2. *Shuffled One-wise*: Shuffles the elements from A and B before performing a one-wise combination;
3. *Index of Each*: Selects elements in a given position (index) from A and B , or the highest position if the position does not exist in a set;
4. *Single Random*: Randomly selects a single element from A and B ;
5. *Pair-wise*: Performs 2-way combinations of the elements from A and B ;
6. *Cartesian Product*: Performs all the possible combinations from A and B .

Other strategies (*e.g.*, shuffled pairwise, t -wise, shuffled t -wise) can be added in the future. Comparisons of these strategies are out of this thesis' scope.

Approaches that use *pseudo-random selection* of elements for combination, such as the Shuffled One-wise or the Single Random, consider an input **random seed**. The sequence of numbers produced by a pseudo-random algorithm varies according to its random seed, that is, whether we always use the same seed, the same sequence of numbers will be produced. **All the strategies and algorithms included in this thesis' approach considers a *unique* input random seed. By default, that seed receives the current date and time.** A user can change it if needed. If the random seed is the same, all the algorithms produce the same results. This behavior makes our approach *predictable* and its outcomes *reproducible*, when needed. However, since we want to use random-based approaches to cover paths or combinations *over time*, our approach always produces a new random seed –unless a seed is given.

7.3.4. Test scenario generation

The generation of Test Scenarios is based on the combination of States (see 6.1.11). A Variant, *B*, can produce a State by declaring it in a Then sentence. For example, the sentence “Then I have the ~item added to the cart~” makes a Variant to produce the state “item added to the cart”. A Variant, *A*, can require or call a Variant *B* by declaring a Given sentence, or a When sentence, that references a State produced by *B*. For example, both the sentences “Given that I have the ~item added to the cart~” and “When I have the ~item added to the cart~” requires the state “item added to the cart”. To create a Test Scenario, the approach replaces steps that call States with their producers, and removes steps that produce States. Figure 15 illustrates two features, A and B, and Test Scenarios created for the Variant VB1 (from B). Since VB1 requires the state “State 1”, which is both produced by the Variants VA1 and VA2 (from A), two Test Scenarios are produced: one that combines VA1 and VB1, and other that combines VA2 and VB1.

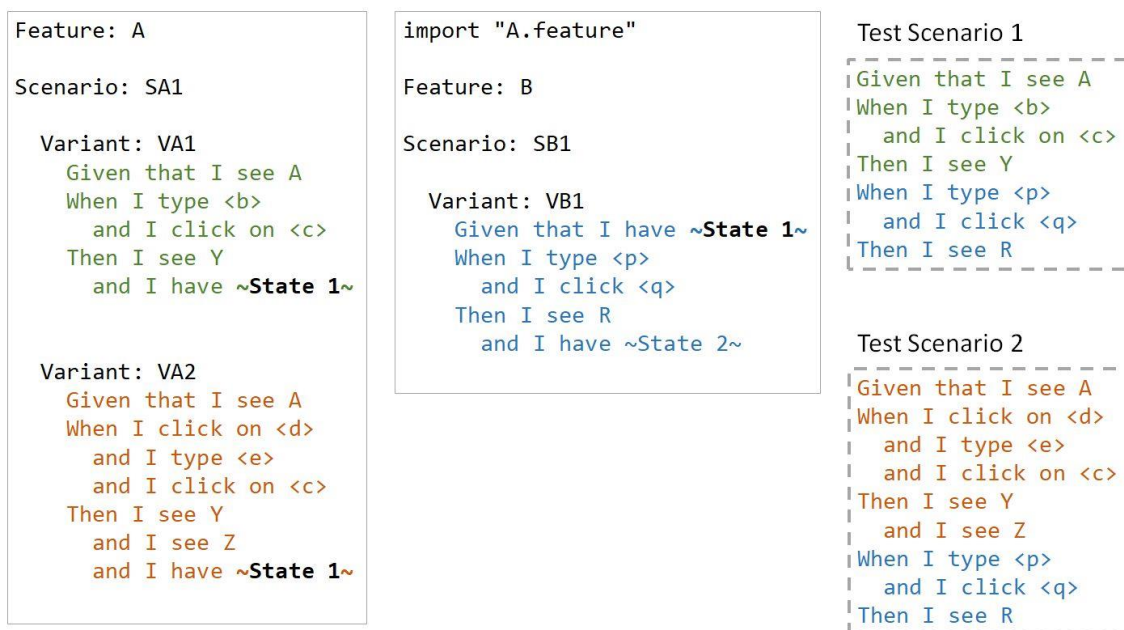


Figure 15 - Example of test scenarios

We defined **minimization strategies** to reduce the number of produced Test Scenarios. It concerns with the selection and the combination of Variant, considering their produced States. When a Variant *requires* a certain a State, the approach searches in the current Feature and in the imported Features for all the Variants that

produces that State (e.g., all the Variants that declare the state “item added to the cart”). However, different Variants may produce a same State. *Thus, we may replace a sentence that requires a State with many different State producers.* This problem becomes more prominent when we consider that a same Variant can require many States. It becomes even bigger when we remember that a same State can be produced by many Variants. Combining them using *Cartesian product* can be unfeasible for many real-world applications. Hence, we define better strategies to select the Variants that produce these States and to combine them.

The algorithm uses *topological sort* to avoid starting from Features and Variants that have dependencies. It sorts the specified Features according to their Imports clauses, then sorts their Variants according to declared States. After that, it transforms every Variant into a Test Scenario. A Test Scenario has the same structure as a Variant, so the strategies above apply in the same way.

7.3.4.1. Variant selection and combination strategies

When a Variant requires a State – for example, it declares a sentence like “Given that I have ~logged user~” to require the state “logged user” – the approach must search it in the current Feature and in the imported Features for all the Variants that produces that State (e.g., all the Variants that has a Then sentence which declares the state “logged user”). Using one of the following strategies can reduce the number of Variants to combine:

1. *First Variant*: always select the first Variant;
2. *Single Random*: selects a random Variant;
3. *First Most Important*: selects the first variant among the most important ones, according to the tag @importance.
4. *All Variants*: does not minimize the selection.

By default, the approach adopts the *Single Random* strategy. The idea is covering all possible combinations *over time*.

To combine all the selected Variants, the approach may use one of the combination strategies defined in section 7.3.3. By default, it uses the *Single Random* strategy, aiming to cover possible combinations *over time*.

7.3.5. Test data generation

Our approach combines well-known techniques to discover defects: equivalence partitioning classes, limit-value analysis, and random generation (BEIZER, 2003; MYERS; THOMAS & SANDLER, 2011). Table 13 presents the *data test cases* included in our approach.⁶³

Table 13 - Data test cases

#	Identification	Description (produces...)
1	VALUE_LOWEST	The lowest value applicable to the data type of the UI Element
2	VALUE_RANDOM_BELOW_MIN	A random value below the specified minimum value
3	VALUE_JUST_BELOW_MIN	The value exactly below the specified minimum value
4	VALUE_MIN	The minimum value
5	VALUE_JUST_ABOVE_MIN	The value exactly above the specified minimum value
6	VALUE_ZERO	Zero (0)
7	VALUE_MEDIAN	The median between the specified minimum and maximum values
8	VALUE_RANDOM_BETWEEN_MIN_MAX	A random value between the specified minimum and maximum values
9	VALUE_JUST_BELOW_MAX	The value exactly below the specified maximum value
10	VALUE_MAX	The maximum value

⁶³ Only the latter two data test cases from Table 13, about computation, were not included the prototype tool. However, a user can simulate the expected outputs of computations directly in the specification (e.g., in a Table or in a sentence).

#	Identification	Description (produces...)
11	VALUE_JUST_ABOVE_MAX	The value exactly above the specified maximum value
12	VALUE_RANDOM_ABOVE_MAX	A random value above the specified maximum value
13	VALUE_GREATEST	The greatest value applicable to the data type of the UI Element
14	LENGTH_LOWEST	A string with length zero
15	LENGTH_RANDOM_BELOW_MIN	A string with a random length below the specified minimum length
16	LENGTH_JUST_BELOW_MIN	A string with the length exactly below the specified minimum length
17	LENGTH_MIN	A string with the specified minimum length
18	LENGTH_JUST_ABOVE_MIN	A string with the length exactly above the specified minimum length
19	LENGTH_MEDIAN	A string whose length is the median between the specified minimum and maximum lengths
20	LENGTH_RANDOM_BETWEEN_MIN_MAX	A string with a random length between the specified minimum and maximum lengths
21	LENGTH_JUST_BELOW_MAX	A string with the length exactly below the specified maximum length
22	LENGTH_MAX	A string with the specified maximum length
23	LENGTH_JUST_ABOVE_MAX	A string with the length exactly above the specified maximum length
24	LENGTH_RANDOM_ABOVE_MAX	A string with a random length above the specified maximum length
25	LENGTH_GREATEST	A string with the greatest length applicable
26	FORMAT_VALID	A value with a valid format, according to the specified format
27	FORMAT_INVALID	A value with an invalid format, according to the specified format

#	Identification	Description (produces...)
28	SET_FIRST_ELEMENT	The first value from the defined set of possible values
29	SET_RANDOM_ELEMENT	A random value from the defined set of possible values
30	SET_LAST_ELEMENT	The last value from the defined set of possible values
31	SET_NOT_IN_SET	A value that is not contained in the defined set of possible values
32	REQUIRED_FILLED	A random valid value
33	REQUIRED_NOT_FILLED	An empty value
34	COMPUTATION_RIGHT	A value produced by a given algorithm
35	COMPUTATION_WRONG	A value that is not produced by a given algorithm

The data test cases apply to a single **UI Element**, and their production varies according to the declared UI Element properties. **UI Literals** always receive *pseudo-random values*, *i.e.*, the data test cases do not apply to them.

Since there are conflicting UI Element properties, it is *not* possible to apply all the data test cases for a single UI Element. Furthermore, some data test cases are not compatible with certain data types, which further reduces the number of applicable data test cases. Table 14 shows UI Element properties and their related data test cases. Table 15 presents the compatibility between UI Element properties and data types. Finally, Table 16 shows the compatibility among UI Element properties.

Table 14 - Data test cases added according to declared properties

Property	Added data test case	Added
<None> or Required	REQUIRED_FILLED	2
	REQUIRED_NOT_FILLED	
Minimum value	VALUE_LOWEST	5
	VALUE_RANDOM_BELOW_MIN	
	VALUE_JUST_BELOW_MIN	
	VALUE_MIN	
	VALUE_JUST_ABOVE_MIN	
Maximum value	VALUE_JUST_BELOW_MAX	5
	VALUE_MAX	
	VALUE_JUST_ABOVE_MAX	
	VALUE_RANDOM_ABOVE_MAX	
	VALUE_GREATEST	
Minimum value + maximum value	VALUE_ZERO	3
	VALUE_MEDIAN	
	VALUE_RANDOM_BETWEEN_MIN_MAX	
Minimum length	LENGTH_LOWEST	5
	LENGTH_RANDOM_BELOW_MIN	
	LENGTH_JUST_BELOW_MIN	
	LENGTH_MIN	
	LENGTH_JUST_ABOVE_MIN	
Maximum length	LENGTH_JUST_BELOW_MAX	5
	LENGTH_MAX	
	LENGTH_JUST_ABOVE_MAX	
	LENGTH_RANDOM_ABOVE_MAX	
	LENGTH_GREATEST	
Minimum length + maximum length	LENGTH_MEDIAN	2
	LENGTH_RANDOM_BETWEEN_MIN_MAX	
Value is/is not/in/not in	SET_FIRST_ELEMENT	4
	SET_RANDOM_ELEMENT	
	SET_LAST_ELEMENT	
	SET_NOT_IN_SET	
Format	FORMAT_VALID	2
	FORMAT_INVALID	
	REQUIRED_NOT_FILLED	

Property	Added data test case	Added
Computed by	COMPUTATION_RIGHT	2
	COMPUTATION_WRONG	

Table 15 - Compatibility between properties and data types

Group	Property	<i>string</i>	<i>integer</i>	<i>double</i>	<i>date</i>	<i>time</i>	<i>datetime</i>	<i>boolean</i>
VALUE	<i>minimum value</i>	×	✓	✓	✓	✓	✓	×
	<i>maximum value</i>	×	✓	✓	✓	✓	✓	×
LENGTH	<i>minimum length</i>	✓	×	×	×	×	×	×
	<i>maximum length</i>	✓	×	×	×	×	×	×
REQUIRED	<i>required</i>	✓	✓	✓	✓	✓	✓	×
FORMAT	<i>format</i>	✓	✓	✓	✓	✓	✓	×
SET	<i>value is/is not/in/not in</i>	✓	✓	✓	✓	✓	✓	×
COMPUTATION	<i>computed by</i>	✓	✓	✓	✓	✓	✓	✓

Table 16 - UI Element property compatibility

Property	<i>min. value</i>	<i>max. value</i>	<i>min.length</i>	<i>max.length</i>	<i>required</i>	<i>format</i>	<i>value is</i>	<i>value is not</i>	<i>value in</i>	<i>value not in</i>	<i>computed by</i>
<i>min. value</i>	-	✓	×	×	✓	✓	×	×	×	×	×
<i>max. value</i>	✓	-	×	×	✓	✓	×	×	×	×	×
<i>min. length</i>	×	×	-	✓	✓	×	×	✓	×	✓	×
<i>max. length</i>	×	×	✓	-	✓	×	×	✓	×	✓	×
<i>required</i>	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓
<i>format</i>	✓	✓	×	×	✓	-	×	✓	×	✓	×
<i>value is</i>	×	×	×	×	✓	×	-	×	×	×	×
<i>value is not</i>	×	×	✓	✓	✓	✓	×	-	✓	×	×
<i>value in</i>	×	×	×	×	✓	×	×	✓	-	×	×
<i>value not in</i>	×	×	✓	✓	✓	✓	×	×	×	-	×
<i>computed by</i>	×	×	×	×	✓	×	×	×	×	×	-

The algorithm for analyzing compatible data test cases for a certain UI Element have to consider the compatibilities from Table 15 and Table 16, as well to consider the applicable data test cases, according to Table 14.

We also defined **strategies to mix data test cases**, *i.e.*, strategies for choosing the mix of data test cases that produces values considered invalid or considered valid. For example, suppose that you are building a desktop application that validates an input data at a time (*e.g.*, by showing the corresponding message). Irrespective whether the approach generates two invalid input data or not, the application will only perform one validation at a time, and, hence, the produced oracles (for both the input data) may not reflect the desired behavior (single oracle). In this way, users can choose the strategy that fits better their applications.

Table 17 presents the strategies to mix data test cases. By default, we adopt the *UnfilteredMix*.

Table 17 - Strategies to mix data test cases

Identification	Number of data test cases that produce invalid values
<i>OnlyValidMix</i>	0
<i>JustOneInvalidMix</i>	1
<i>InvalidPairMix</i>	2
<i>InvalidTripletMix</i>	3
<i>OnlyInvalidMix</i>	all of them
<i>UnfilteredMix</i>	varies, since it leaves as is

Another important task is minimizing the combination of data test cases. Since a single UI Element can have many data test cases, a Test Scenario that has many UI Elements can culminate in a combinatory explosion. For that reason, we can use one of the *combination strategies* defined in section 7.3.3. By default, our approach uses the Shuffled One-wise strategy, that includes every data test case at least once and selects different combinations every time it runs – considering the use of different random seeds. That strategy reduces the number of combinations for frequent tests substantially and allows to cover all the possible combinations (*i.e.*, the Cartesian product) over time.

7.3.6. Test oracle generation

Oracles need to consider the effect of the selected test data in order to establish whether they are *valid* or not. For example, the data test case `VALUE_ZERO` is considered valid only if its UI Element has a minimum value less than or equal to zero, or if it has a maximum value greater than or equal to zero. Whether we can determine the validity of test data, we can produce or use the oracles that correspond to the expected behavior.

UI Element properties can define constraints about the input data. They can also define Otherwise sentences to determine the expected behavior for when these input data do not satisfy the constraints (see section 6.1.10.1). Thus, if we can determine whether an input data satisfy the constraints of a UI Element, then we can determine its validity and the *path* that must be followed. For example, suppose that we can determine that zero (0) is considered an invalid input value for a certain UI Element (based on its minimum and maximum values), and the UI Element contains Otherwise sentences that describe what should happen when the respective constraint is not satisfied. In this case, we can produce a test case that uses the invalid input value and replaces the original Variant postconditions (*i.e.*, Then sentences) by those Otherwise sentences. If zero would be considered valid, we could keep the postconditions in the test case.

In Concordia, therefore, Then sentences and Otherwise are used as oracles. Otherwise sentences usually specify *error handling* behaviors, and they can replace Then sentences when we want to simulate invalid inputs.

7.3.7. Test case generation

A test case is a combination of a test scenario, data test cases (one per UI Element) and test oracles, produced according to the processes and strategies detailed in the previous sections. Figure 16 illustrates how the test case generation process works.

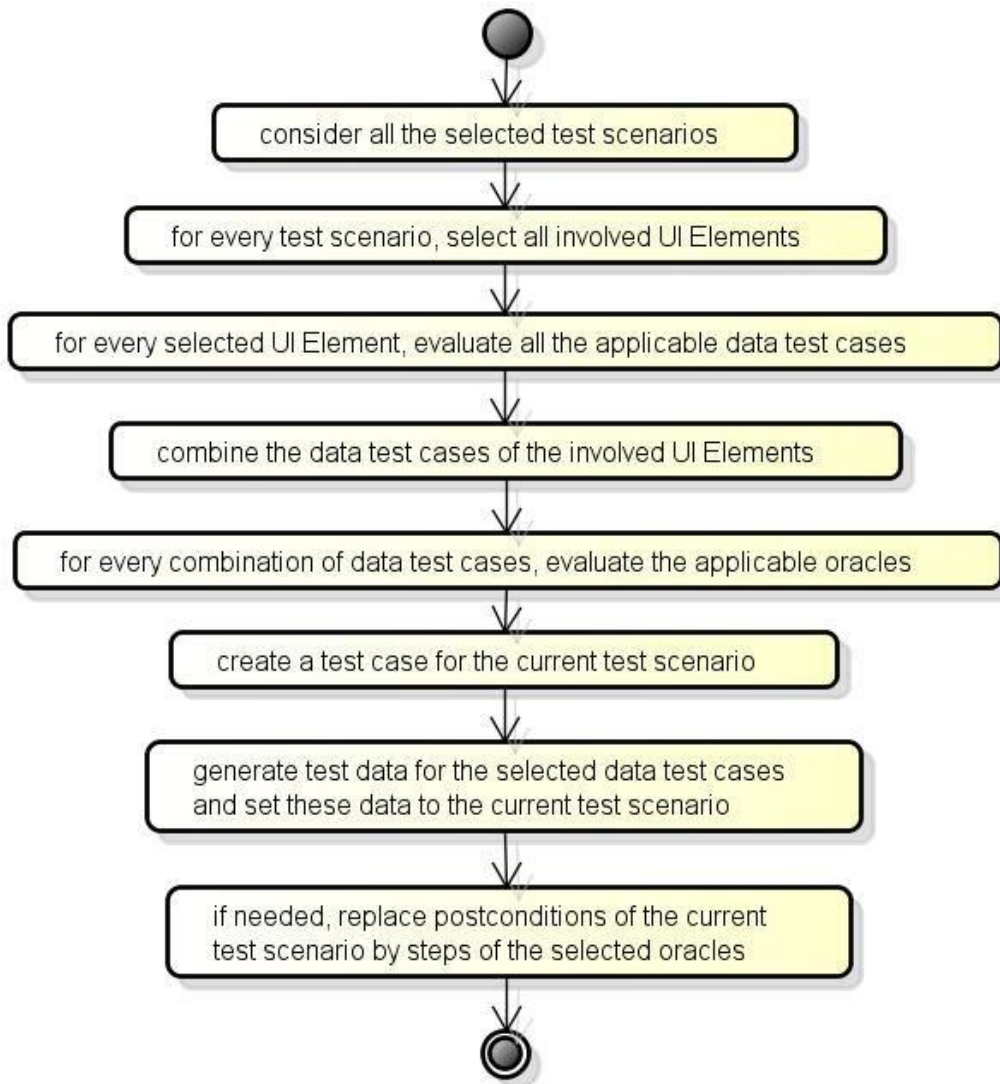


Figure 16 - Test case generation process

The algorithm for producing the Test Cases must adjust Given-When-Then sentences consistently, to keep them in line with the DSL. The algorithm can use the dictionary (see 7.2) for modifying the beginning of the sentences. The same applies when values need to be added to sentences. For example, a sentence like “Given that I fill <age>” that does not have a value, must receive the preposition “with” – according to the dictionary – plus the generated value (and needed spaces): “Given that I fill <age> with 27”.

To help to track their origin, (generated) Test Case names are composed of a Variant name plus an incremental number – e.g., “Logouts by pressing Esc – 1”. Likewise, comments with relevant information are added to Test Case sentences.

These comments can include the origin of values or UI Literals. For example, whether the sentence “When I fill <price> with 100” had the value “100” produced from a Constant named “Min Price”, the comment includes that name, e.g., “When I fill <price> with 100 # [Min Price]”. Whether the value was produced from a Data Test Case named “minimum value”, the comment includes that name, e.g., “When I fill <price> with 100 # minimum value”. Whether the UI Literal was produced from a UI Element named “Sale Price”, the comment also includes it, e.g., “When I fill <price> with 100 # {Sale Price}, minimum value”.

7.3.8. Test script generation, execution, and analysis

Test Cases are converted into Abstract Test Scripts, aiming to have a simple format for transformation into source code. An Abstract Test Script (ATS) needs to have the name of its Feature, Scenario, Variant, and Test Case. *Entities* resulting from Intent Recognition (see 5.6) are used to produce ATS sentences. These sentences are structured as follows:

- *Action*: the action performed in a sentence (usually a verb). For example, “see” is the action of the sentence “Then I see <X>”;
- *Action Modifiers*: modifies the action of the sentence (usually an adverb of negation). For example, “not” is the action modifier of the sentence “Then I do not see <X>”;
- *Action Options*: Adds information to an action. For example, “disabled” is the option of the sentence “Then I see that <X> is disabled”; “seconds” is the option of the sentence “When I wait for 2 seconds”;
- *Targets*: the targets are the involved UI Literals – that is, identifications of the involved UI Elements. For example, “x” is the target of the sentence “Then I see <x>”; “a” and “b” are the targets of the sentence “When I drag <a> to ”;
- *Values*: the values involved in the sentence. For example, “100” is the value of the sentence “When I fill <price> with 100”; “Bob” is the value of the sentence “When I fill <name> with “Bob””;
- *Comment*: the comment retrieved from the Test Case sentence;
- *Location*: the line and column of the sentence in its file.

A tool implemented according to our approach can adopt a plug-in structure to produce test scripts – *e.g.*, that described in Appendix C – in order to not be tied to a specific testing framework. A plug-in must convert Abstract Test Scripts into source code. These ATS contain the needed data (for most testing frameworks, we suppose) to generate (GUI-based) functional test scripts. A plug-in must also be able to run the produced test scripts. For that purpose, it may adopt default configurations or use the configurations produced during its setup process. Executed test scripts can produce output files (*e.g.*, JSON file or XML file), which the plug-in reads for converting to the format expected by the tool.

Execution results must report whether the Test Scripts passed, failed or had errors. A report must consider differences between results expected by Test Cases and results obtained from Test Scripts. These differences should be reported with their cause and their locations in the corresponding files.

7.4.Validation

The aim of a requirements validation is to certify that specified requirements conform to the needs and desires of their stakeholders and they are complete (or complete enough for the intended scope), consistent with standards, not conflicting, not ambiguous, and do not contain technical errors (MAALEM & ZAROOR, 2016). Yousuf *et al.* (2008) point out that the most common techniques for validating requirements are:

- a) *Inspection*: formal evaluation by a group of authors to detect faults or infringements of standards in software requirement documents ;
- b) *reviews*: multiple readers check for omissions and anomalies in requirement documents ;
- c) *prototyping*: an operational model of the application created for discussing and clarifying particular problems of the specification ;
- d) *animation*: walks through specification fragments in order to follow some scenario ;

- e) *language paraphrasing*: a technique which has been devised to tackle the problem caused by two conflicting concerns – the concern of the analyst to develop a formal requirements model, and the users’ need to communicate their requirements in their own universal, widespread terminology, and
- f) *expert systems*: use automated tools provided with domain knowledge to assist the validation of requirements.

Table 18 presents roles and competencies of stakeholders in requirements validation, according to Sommerville (2011). Investing in the collaborative work among all these stakeholders have been considered a good practice to achieve better results in requirements validation. Inayat *et al.* (2015), for example, present a systematic review on agile requirements engineering practices in which they conclude that agile RE practices like customer involvement, review meetings and sessions, changes in requirement management, and cross-functional teams are distinct features missing the traditional way of dealing with requirements. They also affirm that such agile RE practices can outperform and remove the impediments of traditional RE practices, and improve the quality and success rate of outcomes. This strengthens our belief that the proposed process (section 7.1) should be performed with collaborative work for achieving better results.

Although we recommend the collaborative work among the software team, and between stakeholders and the software team, we are neither supposing the practice in studied companies (chapter 9) nor evaluating its impact for validating requirements specifications. The reader may encounter more information about collaborative work in the book by Adzic (2011). He affirms having interviewed 30 teams that implemented around 50 software projects and collaborative work was among the common practices of the most successful teams.

Table 18 - Roles and competences in requirements validation

Stakeholder	Intervention	Roles	Competences and expertise
Analyst	Complete process	Prepares the meetings and ensures the conduct of business objectives	Analysis of IS, animation, and communication, order, decision, negotiation
Customer	Validation	Identify needs read the requirements to verify the correspondence with needs.	Communication
Managers project	Inspection	Use of specifications to plan supply and the development process of the system	Problem domain management cost, delay, technical communication
Domain experts	Validation	Identify Functional requirements	Domain problems and solutions communication
End user	Validation	Spread Functional and non functional requirements, organization, context, constraint	Domain problems and solutions for computers
System engineers and developers	Verification	Use requirements to understand the system under development	Communication HMI
System test engineers	Verification	Use the requirements to develop validation tests for the system	Test enable communication
System maintenance engineers	Maintenance of the validation	Use requirements to help understand the system	Communication
Designers	Verification	Detail and complete the requirements	Communication solution domain

Our approach for validation concerns with two questions :

- **Q1:** *Can Concordia be successfully used for validating requirements with stakeholders?*
- **Q2:** *Can we detect problems in the specification by checking it statically?*

Naturally, the first question is affected by the *easiness* to read and understand the specification – including produced test cases that may serve as examples for validation. We investigate this question in the study detailed in chapter 9.

The second question is affected by algorithms or techniques used to verify declarations in Concordia specifications. We detail these verifications in the following sections, exemplify them in section 8.2, and evaluate the perceptions of users (about them) in chapter 9.

7.4.1.Pre-test defect removal

Table 19 shows the defect removal efficiency of pre-test activities, according to Capers Jones (2014). Its values are similar to those presented by McConnell (2004). Static analysis has high efficiency and its combination with tests can achieve higher defect removal levels, compared to other combinations (JONES, 1996, 2014).

Table 19 – Pre-test defect removal efficiency

Pre-test defect removal	Minimum	Average	Maximum
Formal design inspections	65%	87%	97%
Formal code inspections	60%	85%	96%
<i>Static analysis</i>	65%	85%	95%
Formal requirement inspections	50%	78%	90%
Informal peer reviews	35%	50%	65%
Scrum sessions	35%	55%	70%
Desk checking	25%	45%	55%
Average	49%	69%	80%

In this context, our contributions are: (a) providing a list of verifications for the automatic static analysis of Concordia specifications – which includes verifications for commonly used Agile DSLs (section 4.1); and (b) providing a prototype tool that performs these verifications. The list is presented in Appendix C.

Additionally, we recommend that software teams conduct *informal peer reviews*. They can detect problems such as ambiguity, imprecision, incompleteness, typographical errors, and grammatical errors of feature descriptions and scenario descriptions. Although the software team writes such descriptions using Agile DSL templates (section 4.1), the approach does not consider their sentences for testing purposes and, thus, does not check their syntax. Informal peer reviews can also help to detect incorrect priorities of features, scenarios, and variants.

7.4.2.Static verification

In static verification, developers may undertake inspections, reviews or *static analyzers* to detect errors, omissions, inconsistencies, and deviations from the established standards (SOMMERVILLE, 2011). Literature already addresses inspections and reviews substantially for source code, *e.g.*, program inspection was first established by Fagan at IBM in 1976. Thus, we headed our investigations to the static analysis of Agile DSLs, aiming to find defect classes or properties that can be used to verify documents with such specifications – analogously to those used for static code analysis (*e.g.*, undeclared or uninitialized variables, possible array bound violations, unreachable code, uncalled functions, type mismatches).

We could not find approaches for the automated static analysis of Agile specifications. Most approaches analyzed formal specifications, *e.g.*, (DE ALMEIDA FERREIRA & DA SILVA, 2012; HOLTMANN; MEYER & VON DETTEN, 2011), performed manual analysis and did not established ways to using natural language processing or any other means for that purpose. Publications related to the metalanguages from chapter 4 as well as the approaches from chapter 3 narrowed their investigations to test automation. We also could not find the verifications performed by the analyzed metalanguages in their documentation. Unfortunately, we did not have enough time to read their source code – except for the Gherkin parser. All the verifications performed by the Gherkin parser (which is limited to checking the syntax) were included in our approach.

Rane (2017) – already mentioned in chapter 3 – only exemplifies error handling messages produced by the GUI-based tool that reifies his approach. Only one of the (four) exemplified messages checks the syntax of a declaration: whether the user story uses the format “As a/I want/So that”.

Gaikwad & Joeg (2016) conducted an empirical study about user stories to analyze their correctness. Observed problems were classified in the following categories: (a) Grammatical and typo errors; (b) Big user stories; (c) Action and goal in-

terchanged; (d) Ambiguous user stories; (e) Incorrect acceptance criteria; (f) Incomplete acceptance criteria; (g) Incorrect goal; (h) Incomplete user story; and (i) Incorrect priority. The authors propose a set of practices to mitigate them, *i.e.*, to improving the writing of user stories, such as adopting user role modeling and persona support, using a template for writing user stories, using a scale (1-critical, 2-high, 3-medium, 4-low) to prioritize user stories, using Given-When-Then consistently, and proofreading to detect typos and grammatical errors. They evaluated the proposed practices in two workshops and found that there was a substantial increase in the accuracy of user stories. Both the verifications and analysis were, however, performed manually.

Ernst *et al.* (2014) propose an approach for rewriting agile requirements in a formal language called T1, in order to use a framework called RE-KOMBINE to detect contradictory requirements. Their technique is based on paraconsistent reasoning (a.k.a. paraconsistent logic), and symbols represent sentences and logic conflicts are analyzed. The approach does not try to analyze the specification automatically (*e.g.*, using NLP and then trying to infer contradictions) and (since it deals with a different problem) it does not provide a list of defect classes.

In Appendix C, we present a condensed list of verifications for the Concordia language. We recommend seeing chapter 8 for some examples.

7.4.3.Stakeholders' feedback

After static checking the specification and conducting informal peer reviews, it is probably ready for discussion with stakeholders. Their validation is important to attest whether the software team could capture the business needs and transform these needs into Features, Scenarios, and Variants. Features and Scenarios are discussed from the business point of view, whilst Variants give a good idea of how the system is expected to work. Test Cases produced from Variants can exemplify their behavior with different data. UI Elements capture system rules created from business rules and define how the system should behave in case of the inputs are considered invalid (according to these rules). *Instead of having to define many Variants for error handling, the software team can define a single Variant and let the*

approach create the corresponding Test Cases. Stakeholders may opt to validate the Variants plus the UI Elements or the Test Cases. Validating Variants and UI Elements is probably faster than validating the Test Cases, one by one.

7.5.Maintenance

In this section, we describe basic recommendations that probably facilitate the maintenance of Concordia specifications and test scripts produced from it.

A software team must consider putting all the specification files (extension *.feature*), test case files (extension *.testcase*), and test script files (the file extension varies according to the user plug-in) under *version control*. These files can live along – and evolve – with the source code. Although exact copies of test case files and test script files can be generated using the *same random seed* as before, keeping them under version control facilitates the teamwork, *i.e.*, coworkers would not need to generate the files, and changes are identified more easily by the team.

Nowadays, text editors and IDEs support the syntax highlighting of a plethora of languages, including Gherkin. While they do not yet support Concordia, we recommend taking benefit of their Gherkin support. Syntax highlighting makes the adopted DSLs easier to read and facilitates to encounter defects. Using a grammar checker also helps to find problems in the text.

7.5.1.Variants and UI Elements

Variants are probably one of the parts of the specification that will receive more maintenance. Since their sentences probably receive frequent feedback from customers and other stakeholders, we strongly recommend to substitute some declarations to make them easier to read – and, therefore, easier to validate. Whenever possible, a team should use references to UI Elements instead of using UI Literals. References to UI Elements are much easier to read since they use names (in business language) instead of identifications. They also make the maintenance of identifications simpler, as they provide a single place of change. Another recommenda-

tion is replacing values with Constants when these values become difficult or confusing to read. A Constant replaces a value with its meaning and may facilitate the conversation with stakeholders.

7.5.2. Test cases

In Test-Driven Development, when a defect is discovered a test case is written to simulate it, before any fixing (BECK, 2003). Whether the test case fails, it succeeded in reproducing the defect. The team can then fix the defect and rerun the same test, in order to see if it passes. Other tests are run (as regression tests) to see if the fix introduced defects. When all the test passes, the team gets confident that it was able to remove the defect successfully. Our approach recommends this practice and provides a high-level DSL to describe test cases. Thus, the team can specify a test case to simulate the system behavior that exposed the defect.

Test cases produced *manually* must be placed in different files from those generated – otherwise, they can be overwritten. Whenever the order (index) of Scenarios and Variants are changed (in `.feature` files), such Test Cases must have their tags updated. For example, if there are three Variants in a Scenario and the third one was moved upwards (and becomes the second one), the tag `@Variant(3)` of the manual Test Case should be updated to `@Variant(2)`. The same applies to changes in the order of Scenarios (*i.e.*, it is needed to update the tag `@Scenario`). Test Cases produced automatically do not need these manual updates.

7.5.3. Test scripts

Test scripts generated by a tool that implements our approach should not be edited. Instead, the team should change the corresponding test cases (in `.testcase` files). We also recommend creating additional test scripts in separate files, since the test script generator should always overwrite existing files.

7.6. Concluding remarks

This chapter presented many problems that should be addressed by a *unified* V&V approach for real-world applications. The chapter also detailed how the proposed approach tries to mitigate these problems. The approach defines – based on the current state of the art – an *integrated* set of algorithms, processes, and practices regarding verification, validation, and maintenance. It is the first approach to generate *full-featured* ready to use test cases and test scripts from agile requirements specifications, as well as the first *integrated* approach for V&V of agile requirements specifications.

8 Proof of Concept

Talk is cheap. Show me the code.
- Linus Torvalds

This chapter aims to illustrate the approach's capacity to detect problems in Concordia requirements specifications and to produce tests that can detect differences between such specifications and a system under test.

We built a prototype tool that implements most of the proposed techniques and algorithms for V&V. This prototype was used in the case studies with software companies for receiving feedback. Chapter 9 details these studies. In this chapter, we use the tool for demonstrating some of the approach through examples.

The tool is available at <http://concordialang.org>,⁶⁴ in which there are installation procedures and some documentation about the language and the tool. During the case studies with software companies, we also created a browser plug-in called Katalon-Concordia⁶⁵ that mitigates (and sometimes resolves) the problem related to collecting identifications of user interface elements manually in existent web applications. The plug-in transforms interactions with web applications recorded with a (record-and-playback) software called Katalon Recorder⁶⁶ into sentences in Concordia language. Since it is a complementary tool, we do not demonstrate it here. We edited the exemplified specifications using (the text editor) VS Code.⁶⁷

8.1. Selected cases

To demonstrate the prototype tool's capacity to detecting problems in Concordia specifications, we selected five cases. Table 20 presents them.

⁶⁴ This domain currently redirects to the projects' page, at <https://github.com/thiagodp/concordialang>.

⁶⁵ Available at <https://github.com/thiagodp/katalon-concordia>

⁶⁶ Available at <https://www.katalon.com/resources-center/blog/katalon-automation-recorder/>

⁶⁷ Available at <https://code.visualstudio.com>.

Table 20 – Cases selected to exemplify problems detection

#	Case
1	Invalid Names and Unrecognized Entities
2	Syntax of Actions and the Order of Given-When-Then Sentences
3	Missing States
4	Missing UI Elements and Conflicting Properties
5	Connection with Databases

Table 21 presents the cases selected to exemplify the prototype tool's capacity to produce tests from Concordia specifications and to detect differences between these specifications and a system under test. We used a simple, open source inventory system collected from the Internet to produce the examples.⁶⁸

Table 21 - Cases selected to exemplify the produced tests

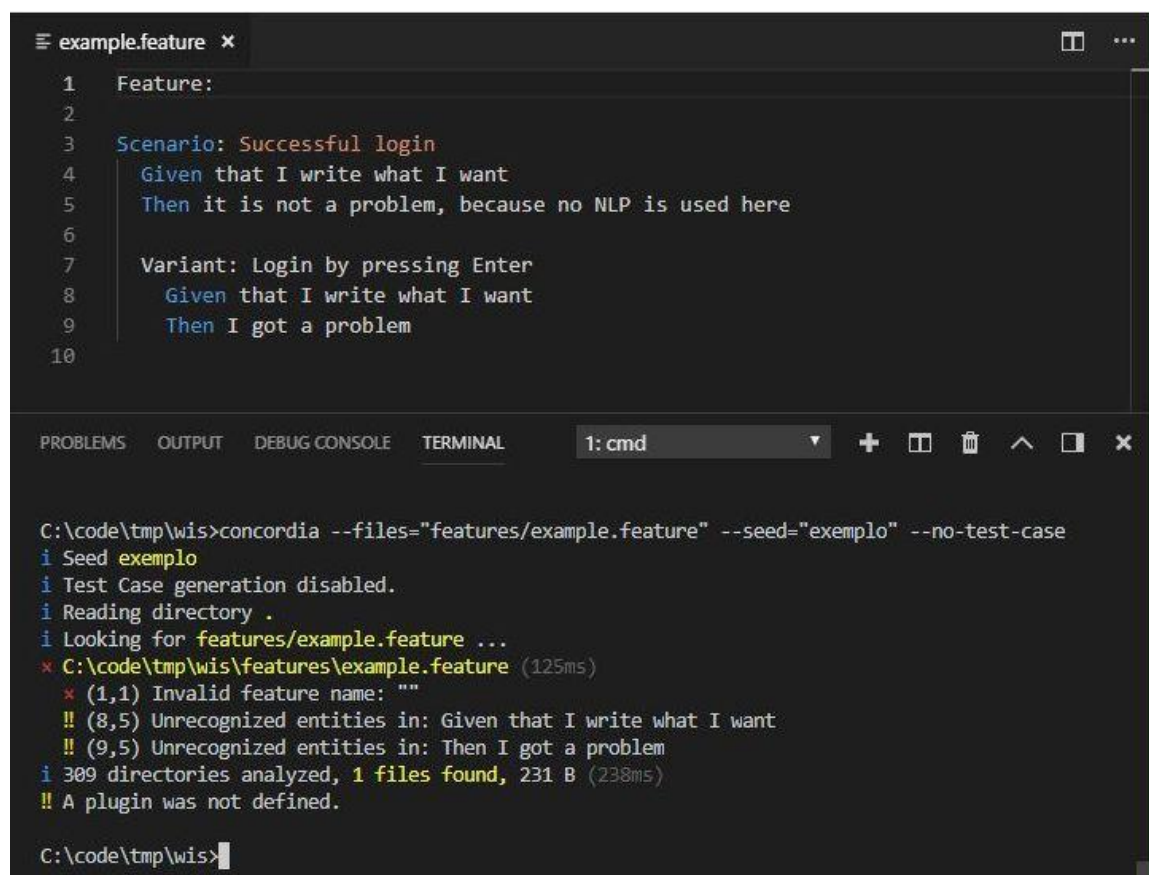
#	Case
1	Testing a Single Feature
2	Testing Related Features
3	Using an External Database
4	Detecting Changes in the System Under Test

⁶⁸ The system was selected due its simplicity for demonstration purposes and its easiness to setup and use. It is available at <https://github.com/siamon123/warehouse-inventory-system>.

8.2. Detecting problems in specifications

Case 1: Invalid Names and Unrecognized Entities

Figure 17 presents an example that checks a Feature name and declared Given-When-Then sentences. Since Scenarios are not used for producing Test Cases, their sentences are not validated using natural language processing. Variants sentences, however, are validated. The NLP processor reports any sentences that it cannot recognize.



```

example.feature x
1 Feature:
2
3 Scenario: Successful login
4   Given that I write what I want
5   Then it is not a problem, because no NLP is used here
6
7 Variant: Login by pressing Enter
8   Given that I write what I want
9   Then I got a problem
10

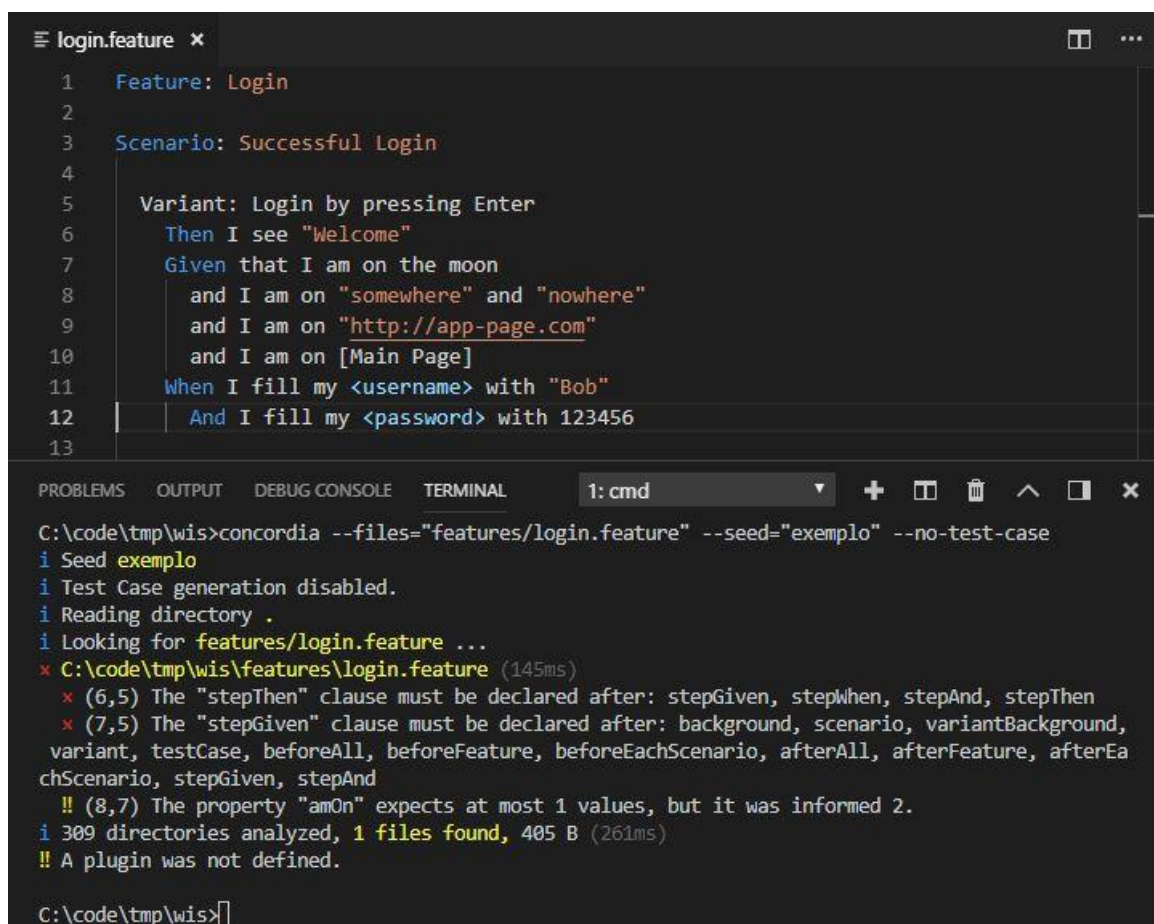
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: cmd
C:\code\tmp\wis>concordia --files="features/example.feature" --seed="exemplo" --no-test-case
i Seed exemplo
i Test Case generation disabled.
i Reading directory .
i Looking for features/example.feature ...
x C:\code\tmp\wis\features\example.feature (125ms)
  x (1,1) Invalid feature name: ""
  !! (8,5) Unrecognized entities in: Given that I write what I want
  !! (9,5) Unrecognized entities in: Then I got a problem
i 309 directories analyzed, 1 files found, 231 B (238ms)
!! A plugin was not defined.

C:\code\tmp\wis>
  
```

Figure 17 - Verification Case 1

Case 2: Syntax of Actions and the Order of Given-When-Then Sentences

Figure 18 illustrates the validation of actions' parameters and the order of Given-When-Then sentences. In the example, the Variant has four Given sentences, all of them containing "am on", which is recognized as an entity called "amOn", and corresponds to being in a certain web page, URL, screen, or window. That entity is an action that requires at least 1 value and at most 1 value. A Constant can be used instead of a value. The NLP processor identifies the syntax correctly. The parser identifies that the Variant starts with a Then sentence, instead of starting with a Given sentence.



```

login.feature x
1 Feature: Login
2
3 Scenario: Successful Login
4
5 Variant: Login by pressing Enter
6   Then I see "Welcome"
7   Given that I am on the moon
8     and I am on "somewhere" and "nowhere"
9     and I am on "http://app-page.com"
10    and I am on [Main Page]
11   When I fill my <username> with "Bob"
12   And I fill my <password> with 123456
13

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: cmd
C:\code\tmp\wis>concordia --files="features/login.feature" --seed="exemplo" --no-test-case
i Seed exemplo
i Test Case generation disabled.
i Reading directory .
i Looking for features/login.feature ...
x C:\code\tmp\wis\features\login.feature (145ms)
  x (6,5) The "stepThen" clause must be declared after: stepGiven, stepWhen, stepAnd, stepThen
  x (7,5) The "stepGiven" clause must be declared after: background, scenario, variantBackground,
variant, testCase, beforeAll, beforeFeature, beforeEachScenario, afterAll, afterFeature, afterEachScenario, stepGiven, stepAnd
  !! (8,7) The property "amOn" expects at most 1 values, but it was informed 2.
i 309 directories analyzed, 1 files found, 405 B (261ms)
!! A plugin was not defined.

C:\code\tmp\wis>

```

Figure 18 - Verification Case 2

Case 3: Missing States

Figure 19 presents a validation of a State. The Variant “Logout by pressing Esc” *requires* the state “logged user”, but that state is not *produced* by the imported file. The tool validates the State and then generates test cases.

The screenshot displays a code editor with two feature files and a terminal window below them.

Left Panel (logout.feature):

```

1  import "login.feature"
2
3  Feature: Logout
4
5  Scenario: Successful logout
6
7      Variant: Logout by pressing Esc
8          Given that I have ~logged user~
9              and I see the {Logout Option}
10             When I press "Escape"
11             Then I do not see the {Logout Option}
12
13  UI Element: Logout Option
14      - type is link
  
```

Right Panel (login.feature):

```

1  Feature: Login
2
3  Scenario: Successful login
4
5  Variant: I do not produce the state
  
```

Terminal Window:

```

C:\code\tmp\wis\features>concordia --seed "example"
i Seed example
i Reading directory .
i 0 directories analyzed, 4 files found, 864 B (383ms)
✓ Generated C:\code\tmp\wis\features\login.testcase
× Generated C:\code\tmp\wis\features\logout.testcase
  × (8,5) The producer of the state "logged user" was not found.
  !! A plugin was not defined.

C:\code\tmp\wis\features>
  
```

Figure 19 - Verification Case 3

Case 4: Missing UI Elements and Conflicting Properties

Figure 20 presents an example with a UI Element called “Password” that was not declared, but it was used in a Variant. Another UI Element called “Username” has two conflicting properties: minimum length is greater than the maximum length. The tool presents the corresponding errors and warnings. Some of these errors are duplicated because they are produced when the tool tries to generate Test Cases, *i.e.*, the test case generator currently evaluates and reports the problems on demand.

The screenshot shows a code editor with two panes. The left pane displays a Gherkin feature file named `login.feature` with the following content:

```

1 Feature: Login
2
3 Scenario: Successful login
4
5   Variant: Login by pressing Enter
6     Given that I am on the [Login Screen]
7     When I fill my {Username} and my {Password}
8       and I press "Enter"
9     Then I see the [Welcome Message]
10      and I have ~logged user~
11
12 Constants:
13   - "Login Screen" is "http://localhost/app/"
14   - "Welcome Message" is "Welcome"
15
16 UI Element: Username
17   - minimum length is 10
18   - maximum length is 5
  
```

The right pane shows the terminal output of the tool execution, which includes the command `C:\code\tmp\wis>concordia --files="features/login.feature" --seed="exemplo"` and the following error messages:

```

i Seed exemplo
i Reading directory .
i Looking for features/login.feature ...
i 309 directories analyzed, 1 files found, 443 B (277ms)
x Generated C:\code\tmp\wis\features\login.testcase
x Referenced UI Elements not found: Password
x Referenced UI Elements not found: Password
x (16,1) C:\code\tmp\wis\features\login.feature: Error generating value for "Login:Username": The minimum value cannot be greater than the maximum value.
!! Could not retrieve a value from {Password} of login.feature (7,5). It will receive an empty value.
!! (7,5) Could not retrieve a literal from {Password}. Generating "password"
!! Could not retrieve a value from {Login:Username} of login.feature (7,5). It will receive an empty value.
!! Could not retrieve a value from {Password} of login.feature (7,5). It will receive an empty value.
!! (7,5) Could not retrieve a literal from {Password}. Generating "password"
!! Could not retrieve a value from {Password} of login.feature (7,5). It will receive an empty value.
!! (7,5) Could not retrieve a literal from {Password}. Generating "password"
!! A plugin was not defined.
  
```

Figure 20 - Verification Case 4

Case 5: Connection with Databases

Figure 21 shows an example of database connection validation. The tool also validates the existence of files used as databases. Warnings appear more than once because different test cases produced from the feature cannot retrieve a value for the UI Element that references the database through a query.

```

login.feature x
1 Feature: Login
2
3 Scenario: Successful login
4
5 Variant: Usual login
6 Given that I am on the [Login Screen]
7 When I fill my {Username} and my {Password}
8 and I fill the {Captcha}
9 and I click on {OK}
10 Then I see the [Welcome Message]
11 and I have ~logged user~
12
13 Constants:
14 - "Login Screen" is "http://localhost/wis/index.php"
15 - "Welcome Message" is "Welcome to OSWA INV"
16
17 UI Element: Username
18
19 UI Element: Password
20
21 UI Element: Captcha
22 - value comes from "SELECT value FROM [TestDB].captchas"
23 Otherwise I see "Invalid captcha."
24
25 UI Element: OK
26 - type is button
27
28 # Unexistent database
29 Database: TestDB
30 - type is "mysql"
31 - name is "testdb"
32 - username "root"
  
```

```

C:\code\tmp\wis>concordia --seed="example"
i Seed example
i Reading directory .
i 309 directories analyzed, 1 files found, 725 B (283ms)
x Generated C:\code\tmp\wis\features\login.testcase
x ER_BAD_DB_ERROR: Unknown database 'testdb'
!! Could not retrieve a value from {Login:Captcha} of login.
feature (8,7). It will receive an empty value.
!! Could not retrieve a value from {Login:Captcha} of login.
feature (8,7). It will receive an empty value.
!! Could not retrieve a value from {Login:Captcha} of login.
feature (8,7). It will receive an empty value.
!! Could not retrieve a value from {Login:Captcha} of login.
feature (8,7). It will receive an empty value.
!! A plugin was not defined.

C:\code\tmp\wis>
  
```

Figure 21 - Verification Case 5

8.3. Generating test cases and test scripts

Since we picked an existing system to exemplify the produced tests, we had to use reverse engineering to produce the exemplified Concordia specifications. *We try to let the examples short in order to focus more on the approach and less on the system.*

Case 1: Single Feature

Figure 22 shows an example with validations performed by a login screen in two different moments (Figure 22a and Figure 22b).

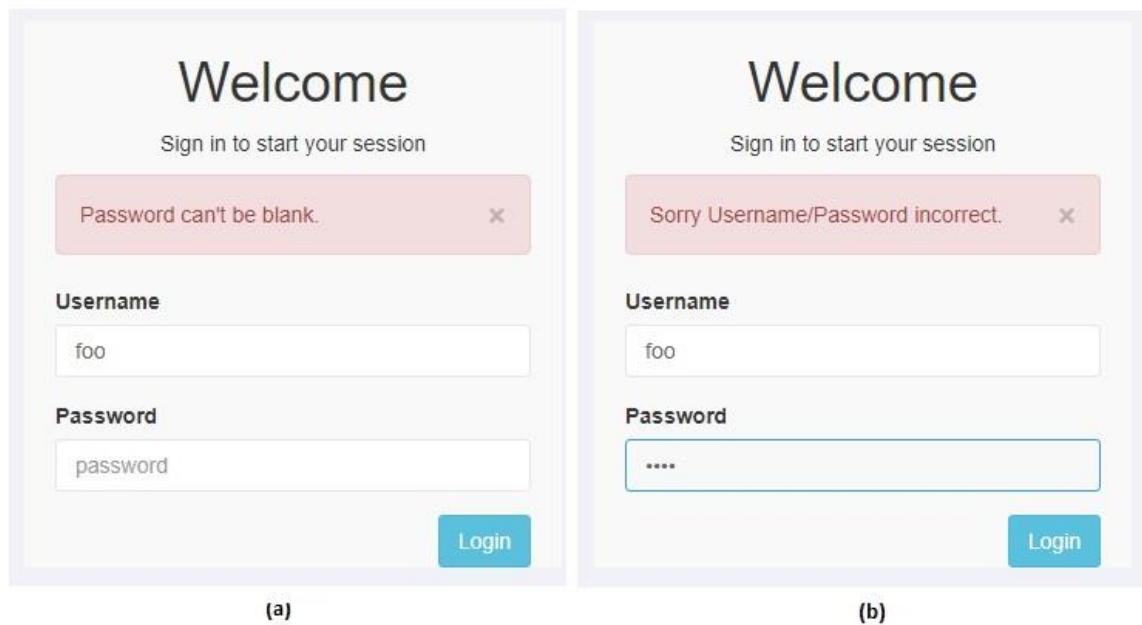


Figure 22 – Some validations in the Login screen

Listing 36 contains the specification created for the login. We defined a table that contains the credentials considered valid, according to the system's documentation. The behavior expected for when the corresponding input is invalid is defined by Otherwise sentences. We defined two simple Variants to illustrate different input possibilities for a same Scenario. Since Concordia produces, by default, camel-cased identifications for UI Elements and the evaluated system also uses this convention, we did not have to define "id" properties in UI Elements.

Feature: Login

As a user

I would like to authenticate myself

In order to access the application

Scenario: Successful login

Given that I can see the login screen

When I enter with valid credentials

Then I can access the application's main screen

Variant: Login by pressing Enter

Given that I am on the [Login Screen]

When I fill my {Username} and my {Password}

and I press "Enter"

Then I see the [Welcome Message]

and I have ~logged user~

Variant: Login by clicking on Login

Given that I am on the [Login Screen]

When I fill my {Username} and my {Password}

and I click on {Login}

Then I see the [Welcome Message]

and I have ~logged user~

Constants:

- "Login Screen" is "http://localhost/wis/index.php"
- "Welcome Message" is "Welcome to OSWA INV"

Table: Users

user	pass	description	
admin	admin	Administrator	
special	special	Special user	
user	user	Default user	

UI Element: Username

- value comes from "SELECT user FROM [Users]"
Otherwise I see the text "Sorry Username/Password incorrect."
- required is true
Otherwise I see "Username can't be blank."

```

UI Element: Password
- value comes from "SELECT pass FROM [Users] WHERE user =
{Username}"
    Otherwise I see the text "Sorry Username/Password incorrect."
- required is true
    Otherwise I see "Password can't be blank."

UI Element: Login
- type is button

```

Listing 36 – Feature Login

Since the feature under test (FUT) presents a single error message at a time, we parameterized the tool to produce a single invalid input value at a time (parameter `--comb-invalid=1`). Thus, the produced oracles for invalid inputs verify a single input at a time. We used the random seed “inventory” to produce the test cases (parameter `--seed="inventory"`) and the plug-in “codeceptjs” (parameter `--plugin="codeceptjs"`) to produce the test scripts.

Listing 37 reproduces **only the test cases generated for the first Variant**. As we explain in section 7.3.7, the approach also produces comments that identify the origin of declarations, data test cases with expectations (valid/invalid), and oracles. The reader may notice that, in every Test Case, the original sentence that contained the action “fill” with two UI Elements was separated in two new sentences and these sentences also received UI Literals (instead of UI Elements) plus the produced test data and comments. For example, in the *fourth* Test Case, the original sentence “When I fill my {Username} and my {Password}” produced the sentences “When i fill <username> with “admin” # {Username}, valid: random element” and “And i fill <password> with “” # {Password}, invalid: not filled”. The produced oracles correspond to the input test data. In that fourth example, the original oracle was changed from “Then I see the [Welcome Message]” to “Then I see “Password can't be blank.” # from <password>”.

```

# Generated with ♥ by Concordia
#
# THIS IS A GENERATED FILE - MODIFICATIONS CAN BE LOST !

import "login.feature"

@generated
@scenario(1)
@variant(1)
Test case: Login by pressing Enter - 1
    Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
    When i fill <username> with "|dj%6j\`çEİf\"ç\'TH®evëTä)ë·İ«ç:\%Ñİñh[5$%¶]=f$Å.k(Q[bδ{C1ÿo2«q\"û-
Íû¢3¢iÅç®;zYª1 #øBúİ¢îD|.,+x%ÎU1u?%³ië@9Y\%(ÃÒ\'·6" # {Username}, invalid: inexistent element
    And i fill <password> with "" # {Password}, valid: last element
    and I press "Enter"
    Then I see the text "Sorry Username/Password incorrect." # from <username>

@generated
@scenario(1)
@variant(1)
Test case: Login by pressing Enter - 2
    Given that I am on the "http://localhost/wis/index.php" # [Login Screen]

```

```

When i fill <username> with "" # {Username}, invalid: not filled
    And i fill <password> with "" # {Password}, valid: first element
    and I press "Enter"
Then I see "Username can't be blank." # from <username>

```

```
@generated
```

```
@scenario(1)
```

```
@variant(1)
```

```
Test case: Login by pressing Enter - 3
```

```
    Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
```

```
    When i fill <username> with "{-ááMEpøaÂ±6(Äé$ \\V,ët;¹@0\%B``Sòò²u^Åe¿²ö\`gúûã³`@Siÿ090d-
    ¤±HQRZLâjç%ârÑ?â ýDF5ã9İÿîwç" # {Username}, invalid: inexistent element

```

```
    And i fill <password> with "" # {Password}, valid: random element
    and I press "Enter"

```

```
    Then I see the text "Sorry Username/Password incorrect." # from <username>

```

```
@generated
```

```
@scenario(1)
```

```
@variant(1)
```

```
Test case: Login by pressing Enter - 4
```

```
    Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
```

```
    When i fill <username> with "admin" # {Username}, valid: random element

```

```
And i fill <password> with "" # {Password}, invalid: not filled
and I press "Enter"
```

```
Then I see "Password can't be blank." # from <password>
```

```
@generated
```

```
@scenario(1)
```

```
@variant(1)
```

```
Test case: Login by pressing Enter - 5
```

```
Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
```

```
When i fill <username> with "admin" # {Username}, valid: first element
```

```
And i fill <password> with "iP¶lgßCè{T\ 'îie-
bYmÁY\>6·@!$%Ûzð¹ôz$ÊÍ-/Eôº)d,úæàÈA_µâßÖê(ãúõ7[L.}-°A%´v%EdgÓûæL×zæ\%OÝÇi±âJæ;ûU. !±¶vÚ·«väBg+äýç2~¿w0%"
# {Password}, invalid: inexistent element
```

```
and I press "Enter"
```

```
Then I see the text "Sorry Username/Password incorrect." # from <password>
```

```
@generated
```

```
@scenario(1)
```

```
@variant(1)
```

```
Test case: Login by pressing Enter - 6
```

```
Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
```

```
When i fill <username> with "user" # {Username}, valid: last element
```

```
And i fill <password> with "" # {Password}, invalid: not filled  
and I press "Enter"
```

```
Then I see "Password can't be blank." # from <password>
```

```
@generated
```

```
@scenario(1)
```

```
@variant(1)
```

```
Test case: Login by pressing Enter - 7
```

```
Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
```

```
When i fill <username> with "admin" # {Username}, valid: first element
```

```
And i fill <password> with "admin" # {Password}, valid: last element  
and I press "Enter"
```

```
Then I see the "Welcome to OSWA INV" # [Welcome Message]
```

```
@generated
```

```
@scenario(1)
```

```
@variant(1)
```

```
Test case: Login by pressing Enter - 8
```

```
Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
```

```
When i fill <username> with "user" # {Username}, valid: last element
```

```
And i fill <password> with "user" # {Password}, valid: random element  
and I press "Enter"
```

```
Then I see the "Welcome to OSWA INV" # [Welcome Message]

@generated
@scenario(1)
@variant(1)
Test case: Login by pressing Enter - 9
  Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
  When i fill <username> with "special" # {Username}, valid: random element
    And i fill <password> with "special" # {Password}, valid: first element
    and I press "Enter"
  Then I see the "Welcome to OSWA INV" # [Welcome Message]
```

Listing 37 - Partial Test Cases produced for Login

In the current version of the prototype tool, the constraint solver does not evaluate the result of queries of related UI Elements to determine whether the produced values are valid or not. For example, because of Password's values are linked to Username's values, an invalid data test case selected for Username makes Password receives an empty value (because the query cannot get a corresponding valid value). The tool produces warnings to make users aware of this and the comments also help users to identify the employed data test cases in case of failing tests. In features or systems that verify a single restriction at a time – as occurs for Login –, this may go unnoticed and not produce failing tests. For example, only when Login receives an invalid input data that Password receives an empty value. Thus, the system criticizes the input for Login and ignores the input for Password.

Listing 38 presents the test scripts that correspond to the test cases from Listing 37. As we informed, these test script were produced for the CodeceptJS testing framework (in JavaScript language). We chose this framework because its methods resemble the syntax of Concordia declarations – *i.e.*, a user can probably notice the correspondence between them. The plug-in for CodeceptJS uses instrumentation to provide *traceability* between test cases and test scripts, *i.e.*, code comments include the corresponding column and line of Concordia declarations, as well as their original comments.

```
// Generated with ♥ by Concordia
// source: c:\code\tmp\wis\features\login.testcase
//
// THIS IS A GENERATED FILE - MODIFICATIONS CAN BE LOST !

Feature("Login");

Scenario("Successful login | Login by pressing Enter - 1", (I) => {
  I.amOnPage("http://localhost/wis/index.php"); // (12,3) [Login Screen]
  I.fillField('username', "íûç3çîÅç®;zYª1 #øBúİçîD|!.+x%ÎU1u?%³ië@9Ý\%(ÃÒ\'·6"); // (13,3) {Username}, invalid: inexistent element
  I.fillField('password', ""); // (14,5) {Password}, valid: last element
  I.pressKey("Enter"); // (15,5)
  I.see("Sorry Username/Password incorrect."); // (16,3) from <username>
});

Scenario("Successful login | Login by pressing Enter - 2", (I) => {
  I.amOnPage("http://localhost/wis/index.php"); // (22,3) [Login Screen]
  I.fillField('username', ""); // (23,3) {Username}, invalid: not filled
  I.fillField('password', ""); // (24,5) {Password}, valid: first element
  I.pressKey("Enter"); // (25,5)
});
```

```

    I.see("Username can't be blank."); // (26,3) from <username>
  });

  Scenario("Successful login | Login by pressing Enter - 3", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (32,3) [Login Screen]
    I.fillField('username',
      "´{-ááMEÞøaÂ±6(Äé$ \\V.ët;¹@%B¨Sòò²u^Åe¿²ö`gúûã³`@Siÿ09@d-
      ¤±HQRZLâj¢%ârÑ?â ýDF5ã9İÿîw¢"); // (33,3) {Username}, invalid: inexistent element
    I.fillField('password', ""); // (34,5) {Password}, valid: random element
    I.pressKey("Enter"); // (35,5)
    I.see("Sorry Username/Password incorrect."); // (36,3) from <username>
  });

  Scenario("Successful login | Login by pressing Enter - 4", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (42,3) [Login Screen]
    I.fillField('username', "admin"); // (43,3) {Username}, valid: random element
    I.fillField('password', ""); // (44,5) {Password}, invalid: not filled
    I.pressKey("Enter"); // (45,5)
    I.see("Password can't be blank."); // (46,3) from <password>
  });

  Scenario("Successful login | Login by pressing Enter - 5", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (52,3) [Login Screen]

```

```

    I.fillField('username', "admin"); // (53,3) {Username}, valid: first element
    I.fillField('password', "íPŋlgßCè{T\ 'Îie-
bYmÁY\>6•@!§\%ÛzĐ¹ôz$ÊÍ-/Eôº)d,úðàÈA_μåßÖê(ãúõ7[L.]~°A%´v%EdgÓûðLxzð\%@ÝÇi±âJð;ûU. !±ŋvÚ•«vâBg+äýç2~¿w@%");
// (54,5) {Password}, invalid: inexistent element
    I.pressKey("Enter"); // (55,5)
    I.see("Sorry Username/Password incorrect."); // (56,3) from <password>
});

Scenario("Successful login | Login by pressing Enter - 6", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (62,3) [Login Screen]
    I.fillField('username', "user"); // (63,3) {Username}, valid: last element
    I.fillField('password', ""); // (64,5) {Password}, invalid: not filled
    I.pressKey("Enter"); // (65,5)
    I.see("Password can't be blank."); // (66,3) from <password>
});

Scenario("Successful login | Login by pressing Enter - 7", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (72,3) [Login Screen]
    I.fillField('username', "admin"); // (73,3) {Username}, valid: first element
    I.fillField('password', "admin"); // (74,5) {Password}, valid: last element
    I.pressKey("Enter"); // (75,5)
    I.see("Welcome to OSWA INV"); // (76,3) [Welcome Message]

```

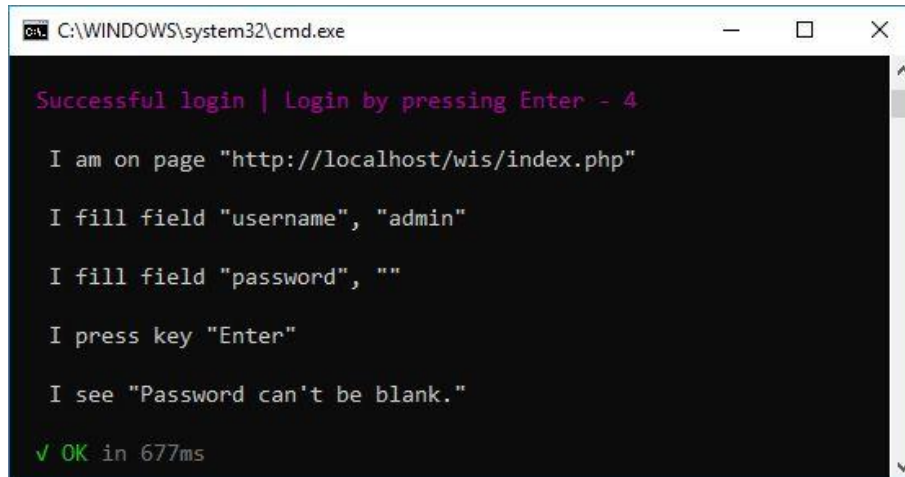
```
});

Scenario("Successful login | Login by pressing Enter - 8", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (82,3) [Login Screen]
    I.fillField('username', "user"); // (83,3) {Username}, valid: last element
    I.fillField('password', "user"); // (84,5) {Password}, valid: random element
    I.pressKey("Enter"); // (85,5)
    I.see("Welcome to OSWA INV"); // (86,3) [Welcome Message]
});

Scenario("Successful login | Login by pressing Enter - 9", (I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (92,3) [Login Screen]
    I.fillField('username', "special"); // (93,3) {Username}, valid: random element
    I.fillField('password', "special"); // (94,5) {Password}, valid: first element
    I.pressKey("Enter"); // (95,5)
    I.see("Welcome to OSWA INV"); // (96,3) [Welcome Message]
});
```

Listing 38 - Partial Test Scripts for Login

Figure 23 illustrates the execution of the fourth test script from Listing 38. When a test script is executed, CodeceptJS produces sentences that resemble sentences in natural language. Therefore, users can follow test executions easily and realize their correspondence to Concordia specifications.



```
C:\WINDOWS\system32\cmd.exe

Successful login | Login by pressing Enter - 4.

I am on page "http://localhost/wis/index.php"

I fill field "username", "admin"

I fill field "password", ""

I press key "Enter"

I see "Password can't be blank."

✓ OK in 677ms
```

Figure 23 - Execution of a test script

Case 2: Testing Related Features

In the system under test, the access to Category depends on the user's level. The user "admin" has access to the menu option "Categories" and can add, edit or remove categories. The user "special" can see the menu option "Categories" but his/her access to categories is denied. Finally, the user "user" cannot see the menu option "Categories". Figure 24 illustrates the system module for Categories.

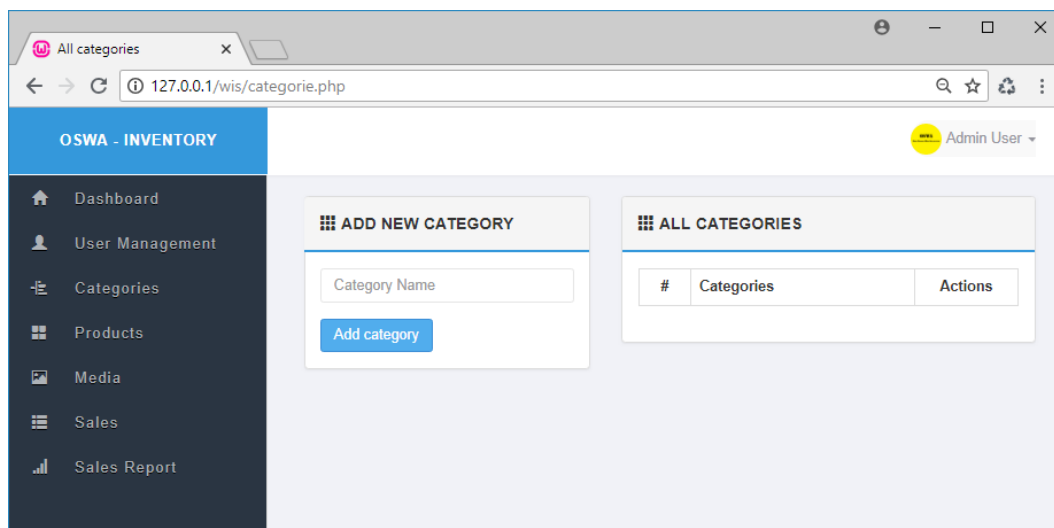


Figure 24 - Access to Categories

Listing 39 - Feature Login modified to consider access rights

Feature: Login

...

Scenario: Successful login

...

Variant: Login by clicking on Login

Given that I am on the [Login Screen]

When I fill my {Username} and my {Password}
and I click on {Login}

Then I see the [Welcome Message]
and I have ~user logged in~

Variant: Administrator Login

Given that I am on the [Login Screen]

When I fill my {Username} with "admin"
and I fill my {Password} with "admin"
and I click on {Login}

Then I see the [Welcome Message]
and I have ~administrator logged in~

Variant: Special User Login

Given that I am on the [Login Screen]

When I fill my {Username} with "special"
and I fill my {Password} with "special"

```

    and I click on {Login}
  Then I see the [Welcome Message]
    and I have ~special user logged in~

Variant: Default User Login
  Given that I am on the [Login Screen]
  When I fill my {Username} with "user"
    and I fill my {Password} with "user"
    and I click on {Login}
  Then I see the [Welcome Message]
    and I have ~default user logged in~

...

```

Listing 40 presents the feature “Add Category” which needs the feature “Login”. Three different Scenarios illustrate the expectations related to the user-level access (explained earlier).

Listing 40 - Feature Add Category

```

import "login.feature"
Feature: Add Category

Scenario: Admin can add category
  Variant: Admin adds category sucessfully
    Given that I have ~administrator logged in~
    When I click "Categories"
      and I fill {Category Name}
      and I click on {Add category}
    Then I see "Successfully Added Category"

Scenario: Special user cannot add category
  Variant: Special user has no permission
    Given that I have ~special user logged in~
    When I click "Categories"
    Then I see "Sorry! you dont have permission to view the page."

```



```

Scenario: Default user cannot add category
  Variant: Default user cannot see the menu Categories
    Given that I have ~default user logged in~
    Then I do not see "Categories"

UI Element: Category Name
  - id is "category-name"
  - required is true
    Otherwise I see "Category name can't be blank."

UI Element: Add category
  - id is "add_cat"

```

For a sake of space, we will only reproduce some of the test cases and test scripts generated from the feature Add Category. Figure 25 shows the first test case generated from that feature. The reader may notice that the steps related to the state “administrator logged in” were replaced by the corresponding steps of the Variant that produces the state. Figure 26 shows the test script that corresponds to the first test case.

```

# Generated with ♥ by Concordia
#
# THIS IS A GENERATED FILE - MODIFICATIONS CAN BE LOST !

import "add-category.feature"

@generated @scenario(1) @variant(1)
Test case: Admin adds successfully - 1
  Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
  When I fill my <username> with "admin" # {Username}
    and I fill my <password> with "admin" # {Password}
    and I click on <Login> # {Login}
  Then I see the "Welcome to OSWA INV" # [Welcome Message]
  When I click "Categories"
    And i fill <category-name> with "cb(i*TBnó1í[0" # {Category Name}, valid: filled
    and I click on <add_cat> # {Add category}
  Then I see "Successfully Added Category"

```

} ~administrator
logged in~

Figure 25 - First Test Case from the Feature Add Category

```
// Generated with ♥ by Concordia
// source: c:\code\tmp\wis\features\add-category.testcase
//
// THIS IS A GENERATED FILE - MODIFICATIONS CAN BE LOST !

Feature("Add Category");

Scenario("Admin can add category | Admin adds sucessfully - 1", (I) => {
  I.amOnPage("http://localhost/wis/index.php"); // (12,3) [Login Screen]
  I.fillField('username', "admin"); // (13,3) {Username}
  I.fillField('password', "admin"); // (14,5) {Password}
  I.click('Login'); // (15,5) {Login}
  I.see("Welcome to OSWA INV"); // (16,3) [Welcome Message]
  I.click("Categories"); // (17,3)
  I.fillField('categorie-name', "cb(i*TBn61i[0]"); // (18,5) {Category Name}, valid: filled
  I.click('add cat'); // (19,5) {Add category}
  I.see("Successfully Added Category"); // (20,3)
});
```

~administrator
logged in~

Figure 26 - First Test Script generated from the Test Case “Admin adds successfully - 1”

Figure 27 shows the third test case generate from the feature Add Category, and Figure 28 shows the corresponding test script.

```
@generated @scenario(2) @variant(1)
Test case: Special user has no permission - 1
  Given that I am on the "http://localhost/wis/index.php" # [Login Screen]
  When I fill my <username> with "special" # {Username}
    and I fill my <password> with "special" # {Password}
    and I click on <Login> # {Login}
  Then I see the "Welcome to OSWA INV" # [Welcome Message]
  When I click "Categories"
  Then I see "Sorry! you dont have permission to view the page."
```

~special user
logged in~

Figure 27 - Third Test Case generated from the Feature Add Category

```

Scenario("Special user cannot add category | Special user has no permission - 1",
(I) => {
    I.amOnPage("http://localhost/wis/index.php"); // (40,3) [Login Screen]
    I.fillField('username', "special"); // (41,3) {Username}
    I.fillField('password', "special"); // (42,5) {Password}
    I.click('login'); // (43,5) {login}
    I.see("Welcome to OSWA INV"); // (44,3) [Welcome Message]
    I.click("Categories"); // (45,3)
    I.see("Sorry! you dont have permission to view the page."); // (46,3)
});

```

Figure 28 - Test Script generated from the Test Case "Special user has no permission - 1"

Case 3: Using an External Database

Databases can be used in UI Element properties and Test Events. These Test Events can configure databases or their execution environments, aiming to have the proper system states when the test scripts run.

In the system under test, Categories are unique and its database reflects that restriction. Whether we execute more than once a test script that adds a certain Category, that test script will fail because of the database restriction. Hence, in order to avoid such test scripts to fail, we can handle the database before or after they run.

Listing 41 shows the content of the file “db.feature” that contains a Database declaration. Listing 42 shows the declaration of the test event Before Feature in the feature Add Category. Listing 43 shows the corresponding source code produced from that test event and Figure 29 shows the execution of a test script that triggers the test event.

Listing 41 - File db.feature

```
Database: WISDB
- type is "mysql"
- host is "localhost"
- name is "oswa_inv"
- username is "root"
- password is ""
```

Listing 42 – Test Event Before Feature declared for the feature Add Category

```
import "db.feature"

...
Before Feature:
  When I connect to the database [WISDB]
    and I run the script 'DELETE FROM [WISDB].`categories`'
```

Listing 43 - Test Script generated from the event Before Feature from Add Category

```
BeforeSuite( async (I) => { // Before Feature
  I.connect("WISDB",
    {"driverName":"mysql",
     "username":"root",
     "password":"",
     "hostname":"localhost",
     "database":"oswa_inv"}
  ); // (38,3)
  await I.run('WISDB', 'DELETE FROM `categories`'); // (39,5)
});
```



```
CodeceptJS v1.3.3
Using test root "c:\code\tmp\wis"

Add Category --

  I connect "WISDB", {"driverName":"mysql","username":"root","password":"","hostname":"localhost","database":"oswa_inv"}
  I run "WISDB", "DELETE FROM `categories`"
Admin can add category | Admin adds category sucessfully - 1

  I am on page "http://localhost/wis/index.php"
  I fill field "username", "admin"
  I fill field "password", "admin"
  I click "Login"
  I see "Welcome to OSWA INV"
  I click "Categories"
  I fill field "categorie-name", "çb(ï*TBnó1i[Ø"
  I click "add_cat"
  I see "Successfully Added Category"

✓ OK in 6433ms
```

Figure 29 - Example of Execution of the Test Event

Case 4: Detecting Changes in the System Under Test

Test scripts that check the system's state from its user interface are sensitive to user interface changes. For example, they can break when UI elements are not found or when the UI content (*e.g.*, values, messages, text) differs from the expectations. **Both Actions and Oracles can break test scripts.** The sensibility of test scripts may also depend on the used testing framework, since they can adopt measures such as *smart wait*, *i.e.*, to give an extra time when waiting for UI elements or values without delaying execution, or *smart search*, *i.e.*, to search for different identifications for a same UI element.

To exemplify the approach's capacity to detect changes in the SUT, we edited the source code of the SUT to remove the constraint that defines the Category's name as a required input field. Listing 44 shows the code (in PHP language) before and after the change. Figure 30 shows the test scripts passing before the change in the SUT. Figure 31 shows that the test script that corresponds to the modified constraint fails after the change in the SUT – that is, it is able to detect the change. Figure 32 shows the details of the failure, which evidence that the expected message used as oracle was not found in the SUT.

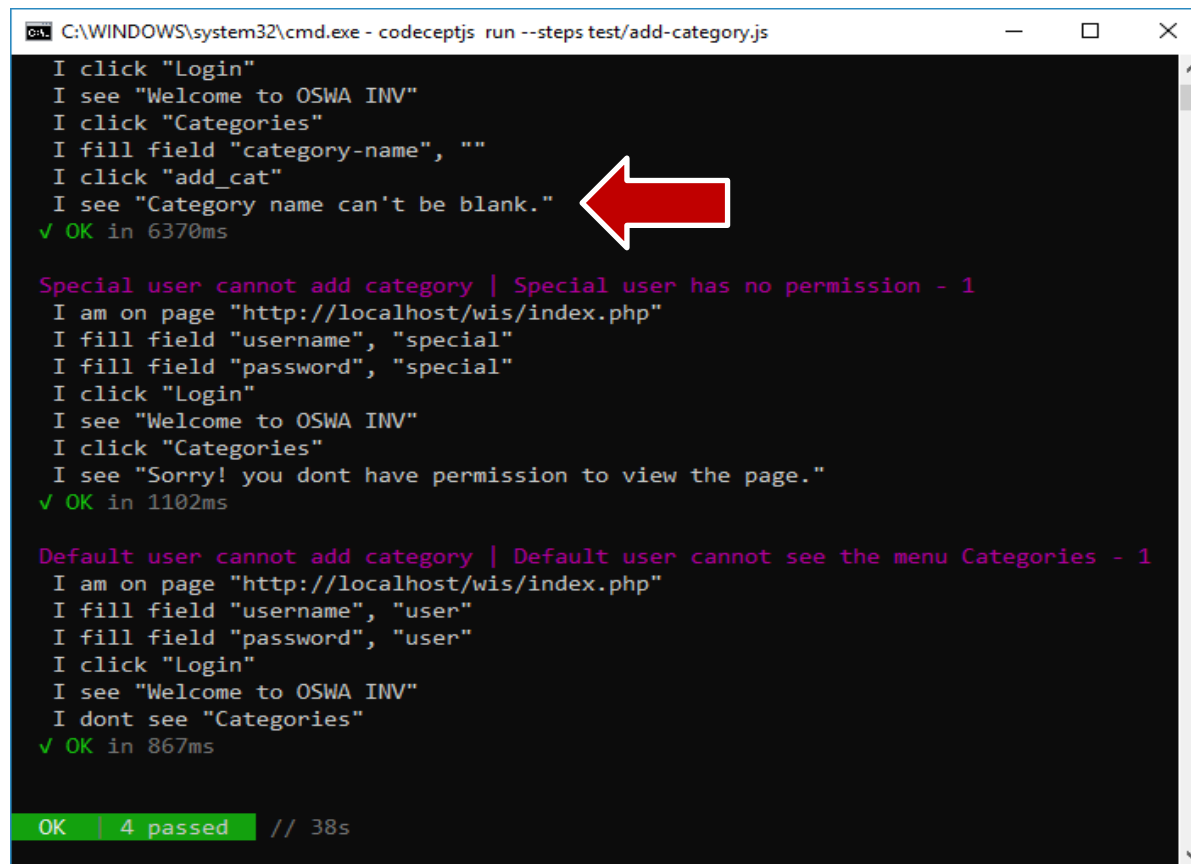
Listing 44 - Change in a required constraint of the SUT

Before:

```
$req_field = array('category-name');  
validate_fields($req_field);
```

After:

```
$req_field = array(); // empty  
validate_fields($req_field);
```



```
C:\WINDOWS\system32\cmd.exe - codeceptjs run --steps test/add-category.js

I click "Login"
I see "Welcome to OSWA INV"
I click "Categories"
I fill field "category-name", ""
I click "add_cat"
I see "Category name can't be blank."
✓ OK in 6370ms

Special user cannot add category | Special user has no permission - 1
I am on page "http://localhost/wis/index.php"
I fill field "username", "special"
I fill field "password", "special"
I click "Login"
I see "Welcome to OSWA INV"
I click "Categories"
I see "Sorry! you dont have permission to view the page."
✓ OK in 1102ms

Default user cannot add category | Default user cannot see the menu Categories - 1
I am on page "http://localhost/wis/index.php"
I fill field "username", "user"
I fill field "password", "user"
I click "Login"
I see "Welcome to OSWA INV"
I dont see "Categories"
✓ OK in 867ms

OK | 4 passed // 38s
```

Figure 30 –Test Script passing before the change in the SUT

```
C:\WINDOWS\system32\cmd.exe - codeceptjs run --steps test/add-category.js

I fill field "password", "admin"
I click "Login"
I see "Welcome to OSWA INV"
I click "Categories"
I fill field "category-name", "çb(ï*TBnó1í[ø"
I click "add_cat"
I see "Successfully Added Category"
✓ OK in 7399ms

Admin can add category | Admin adds category sucessfully - 2
I am on page "http://localhost/wis/index.php"
I fill field "username", "admin"
I fill field "password", "admin"
I click "Login"
I see "Welcome to OSWA INV"
I click "Categories"
I fill field "category-name", ""
I click "add_cat"
I see "Category name can't be blank."
✗ FAILED in 7278ms

Special user cannot add category | Special user has no permission - 1
I am on page "http://localhost/wis/index.php"
I fill field "username", "special"
I fill field "password", "special"
I click "Login"
I see "Welcome to OSWA INV"
I click "Categories"
I see "Sorry! you dont have permission to view the page."
✓ OK in 4064ms
```

Figure 31 - Test Script failing after the change in the SUT


```
C:\WINDOWS\system32\cmd.exe - codeceptjs run --steps test/add-category.js

I dont see "Categories"
✓ OK in 924ms

-- FAILURES:

1) Add Category
  Admin can add category | Admin adds category sucessfully - 2:

  expected web page to include "Category name can't be blank."
  + expected - actual

  -OSWA - INVENTORY
  -August 30, 2018, 11:41 am
  -Admin User
  -Dashboard
  -User Management
  -Categories
  -Products
  -Media
  -Sales
  -Sales Report
  --- ( 8 lines more ) ---
  +Category name can't be blank.

Scenario Steps:

- I.see("Category name can't be blank.") at Object.obj.(anonymous function).obj.(anonymo
nymous function) [as see] (C:\Users\thiag\AppData\Roaming\npm\node_modules\codeceptjs\
```

Figure 32 - Details of the failure

8.4. Concluding remarks

This chapter illustrated the approach's capacity to detect problems in requirements specifications written with Concordia language, the generated test cases and test scripts, and the capacity of these test scripts detecting differences between such specifications and the system under test.

9Multi-case Study

If you're relentlessly focused on lowering cost, you'll quickly become oblivious to opportunities to increase value.

- Michael Bolton

This chapter presents the design of the study, its results, and related conclusions.

We conducted a multi-case study with small and micro software development companies to evaluate the effectiveness of the proposed approach in real software projects. In section 9.1, we present its design. Sections 9.2 and 9.3 show the collected quantitative and qualitative data, respectively. Finally, we discuss the results in section 9.4.

9.1.Study design

9.1.1.Strategy

Case studies are a preferred strategy when the investigator has little control over the events, and when the focus is a contemporary phenomenon within some real-life context (YIN, 2003). Since our approach has a strong relation to the practice, empirical research and observation through case studies fit better its context. Furthermore, multiple-case designs are likely to be stronger than single-case designs (YIN, 2003).

Case studies can be based on any mix of quantitative and qualitative evidence (YIN, 2003). Our research collected quantitative data from the usage of the proposed approach during the study, and at the end of the study through a questionnaire. The qualitative data were collected using a questionnaire and semi-structured interviews. Section 9.1.5 details the procedures and measurements.

9.1.2.Goal and research questions

The goal of our multi-case study is *to evaluate the proposed approach with real-world software projects in relation to its effort and benefits*. More specifically, we are interested in trying to answer the following research questions:

RQ1: How difficult is it for stakeholders to understand requirements specified with Concordia?

RQ2: Is the metalanguage useful to validate requirements with stakeholders?

RQ3: How difficult is it to specify requirements with Concordia?

RQ4: How time-consuming is it to specify requirements with Concordia?**RQ5:** Is the approach useful to verify the compliance of a system with its requirements?

RQ6: How complete are the tests generated with the approach?

RQ7: Is the approach useful to discover defects?

9.1.3.Participants

We offered a free course on functional testing aiming at attracting companies to participate in the study and introducing them to the metalanguage (Concordia) and the prototype tool. The case studies started after the course ended. All the participants took part in the case studies voluntarily.

Table 22 presents companies' profiles, whose names were replaced by letters.

Table 22 - Companies' Profiles

<i>Company</i>	D	O	H	I
<i># of employees</i>	26	10	6	3
<i>development process</i>	Prescriptive, Agile	Prescriptive, Agile	Prescriptive	Agile
<i>types of software</i>	Information systems	Information systems	Information systems	Information systems, mobile applications

<i>evaluated software domains</i>	Sales and content management	Veterinary examinations	Enterprise Resources Planning	Account management (payables and receivables)
<i>Req. spec. language</i>	Portuguese	Portuguese	Portuguese	Portuguese, English

Company “D” was founded in 2008 and its main activities are software development and digital marketing. Its main product is an e-commerce platform and Concordia was mostly used with features of the new version of that platform, which is about to be released.

Company “O” was founded in 2013 and started developing web-based systems in the last three years. It decided to adopt Concordia to continue developing features of its software for veterinary examinations.

Company “H” was founded in 2010 and develops integrated commercial applications, such as enterprise resource planning (ERP) and point-of-sale software. It used Concordia in features of the ERP software, which use web-based technologies.

Company “I” is a software development startup founded in 2017. It used Concordia in its web-based software for managing payables and receivables.

Table 23 summarizes participants’ profiles, whose names were replaced by letters.

Table 23 - Participants' Profiles

#	<i>Id</i>	<i>Company</i>	<i>Current position / time in the co.</i>	<i>Education</i>
1	D-D	<i>D</i>	Developer-Tester, 1 year, 2 months	Technical high school in Informatics, Undergraduate student (6 th semester)
2	D-G		Developer, 1 year, 2 months	Technical high school in Informatics, Undergraduate student (6 th semester)
3	O-A	<i>O</i>	Developer-Tester, 2 years, 7 months	Technical high school in Informatics, Undergraduate student (6 th semester)
4	O-V		Developer, 5 months	Technical high school in Informatics, Bachelor's degree
5	O-R		Developer, 3 months	Technical high school in Informatics
6	H-W	<i>H</i>	Developer/Tester, 7 years	Technical high school in Informatics, Undergraduate student (7 th semester)
7	H-V		Developer/Tester, 3 months	Undergraduate student (10 th semester)
8	I-W	<i>I</i>	Developer/Tester, 3 years, 4 months	Technical high school in Informatics, Undergraduate student (7 th semester)
9	I-S		Developer/Tester, 8 months	Technical high school in Informatics, Undergraduate student (5 th semester)

9.1.4. Initial procedures

After the companies accepted the invitation to participate in the case studies, we scheduled and performed a presentation to their management and employees. We then scheduled a kickstart meeting in which we supervised the specification with Concordia and the tool usage. The companies started specifying simple features and proceeded to more complex ones as they learned. Additional meetings were performed when needed to clarify doubts. We also offered support via e-mail, mobile phone, Skype⁶⁹, and Telegram⁷⁰. The study lasted approximately two months (March-July, 2018) and started immediately after finishing the prototype tool (July 2017 to February 2018) and the functional testing course (March 2018).

⁶⁹ <https://www.skype.com>

⁷⁰ <https://telegram.org>

9.1.5.Procedures and measurements

Basri & O'Connor (2010) point out that, in small companies, most of the management processes are performed through an informal way and less documented, due to a small number of people involved in the project and the organization. During the presentation with management or the kickstart meeting, we identified that the companies did not collect metrics about their processes and they probably would not do (during the case studies) due to different reasons, such as lack of time, company culture, or inappropriate qualification. We then decided to reduce the number of quantitative metrics adopted in the case studies. For example, we did not include *Defect Removal Efficiency* (DRE), a quality metric that can be used to evaluate the capacity of a software team to detect defects in a software after releasing it to customers (SOMMERVILLE, 2011) – in order to evaluate whether the company's efficiency would improve after adopting our approach.

We asked the participants to take notes on: (i) defects found with the prototype tool; (ii) unexpected inconsistencies between the requirements specification and the system under test; and (iii) stakeholders' feedback about specified requirements. We also collected feedback during conversations, meetings and support tasks. At the end of the study, we applied a questionnaire to the participants listed in Table 23. The questionnaire is composed of ranked questions, Likert-scale questions, and open-ended questions. Table 24 presents the questionnaire rationale, constructed using Goal-Question-Metric (GQM) (BASILI, 1992). Finally, we conducted semi-structured interviews with participants to gather additional qualitative feedback (see 9.3.1) and to validate some questionnaire's answers. We also used GQM to formulate the base questions for these interviews, which are available in the Table 25.

Table 24 - Questionnaire rationale

Goal (to evaluate...)	#	Question	Metric
Number of applications involved	1	<i>In how many applications did you use Concordia?</i>	Number of applications

Goal (to evaluate...)	#	Question	Metric
Number of features involved	2	<i>In how many features did you use Concordia?</i>	Number of features
Frequency of validation with the customers	3	<i>Whether you could validate features with stakeholders, every how many days did this validations occur?</i>	Number of days
Types of systems involved	4	<i>In which types of system did you use Concordia? () Information Systems () Websites () Mobile applications () Games () Other</i>	Type of system
Maturity of features	5	<i>Please indicate the number of evaluated features according to the number of months or years they have. Not released yet: ____ Up to 3 months: ____ Up to 6 months: ____ Up to 12 months: ____ From 1 to 3 years: ____ More than 3 years: ____</i>	Number of features per time interval
Participant's background and experience	6	<i>Please check the degrees that correspond to your qualification: [] Elementary school [] Middle school [] High school [] Associate's [] Bachelor's [] Postgraduate's [] Master's [] Doctor's [] Other _____</i>	Academic background
	7	<i>Are you currently attending any course? Please inform the corresponding semesters.</i>	Formation
	8	<i>Please inform which positions you occupied in your company and for how long.</i>	Months by position

Goal (to evaluate...)	#	Question	Metric
	9	<i>Please check the types of requirements specification documents that you have prior experience: [] Wiki or textual documentation [] Use cases [] User Stories [] Other ____</i>	Types of documents
	10	<i>Please check the types of automated tests that you have prior experience. [] Unit or integration tests [] Web API tests [] Functional or UI tests [] Non-functional tests (load/performance/security/other)</i>	Types of automated test
Quality control practices	11	<i>Before using Concordia, which types of requirements specification documents did your company use? [] Wiki or textual documentation [] Use cases [] User Stories [] Other ____</i>	Types of documents
	12	<i>Before using Concordia, how frequent did your company validate requirements with customers? () Never () Very rarely () Rarely () Sometimes () Frequently () Always</i>	Frequency of validation
	13	<i>Before using Concordia, how frequent did your company use requirements specifications for producing tests? () Never () Very rarely () Rarely () Sometimes () Frequently () Always</i>	Frequency of tests based on specifications

Goal (to evaluate...)	#	Question	Metric
	14	<i>Before using Concordia, which types of automated tests did your company use? [] None [] Unit or integration tests [] Web API tests [] Functional or UI tests [] Non-functional tests (load/performance/security/other)</i>	Types of automated test
Language comprehension and usage	15	<i>How hard was for customers to comprehend Concordia specifications? () I could not show them () Very hard () Hard () Neither hard nor easy () Easy () Very easy</i>	Perception of level of difficulty
	16	<i>Please make additional comments about any customers' difficulties to comprehend Concordia specifications.</i>	Observations (descriptive)
	17	<i>Did you provide any explanations for your customers before letting them read Concordia specifications? What explanations?</i>	Observations (descriptive)
	18	<i>Your difficulty to understand Concordia specifications was: () None () Very small () Small () Normal () High () Very high () Total</i>	Perception of level of difficulty
	19	<i>Which parts of the specification were harder to understand?</i>	Observations (descriptive)
	20	<i>Your difficulty to write Concordia specifications was: () None () Very small () Small () Normal () High () Very high () Total</i>	Perception of level of difficulty
	21	<i>Which parts of the specification were harder to write?</i>	Observations (descriptive)

Goal (to evaluate...)	#	Question	Metric
	22	<i>The time invested to write Concordia specifications was: () Very short () Short () Reasonable () Long () Very long</i>	Perception of time
	23	<i>When compared to the time that your company used to invest on specification documents, the time invested to write Concordia specifications was: () The company did not write specifications () Much faster () Faster () Equal () Slower () Much slower</i>	Perception of time
	24	<i>When compared to requirement specification documents previously used in your company, Concordia was: () The company did not write specifications () Much easier () Easier () Equal () Harder () Much harder</i>	Perception of easiness
Produced tests	25	<i>Compared to your company's prior practice to produce tests, Concordia was: () Much easier () Easier () Equal () Harder () Much harder</i>	Perception of easiness
	26	<i>How do you classify the number of tests produced by Concordia, in relation to the practice previously adopted by your company? () Much larger () Larger () Equal () Smaller () Much smaller</i>	Perception of number of tests

Goal (to evaluate...)	#	Question	Metric
	27	<i>How do you classify the quality of tests the produced by Concordia, in relation to the practice previously adopted by your company? () Much higher () Higher () Equal () Lower () Much lower</i>	Perception of quality of tests
Effectiveness	28	<i>How many defects do you remember had discovered with Concordia?</i>	Number of defects
	29	<i>Which other problems in your application do you remember had discovered with Concordia?</i>	Observations (descriptive)
	30	<i>In most of your Variants, you... () Fixed all used test data () Fixed some of the used test data () Let Concordia generate some of the used test data () Let Concordia generate all the used test data</i>	Usage of manually defined test data
	32	<i>In most of your Features, you... () Did not specify UI Elements () Specified some UI Elements but did not specified properties related to their values. () Specified UI Elements with properties related to their values.</i>	Usage of data properties
The capacity of being used for validation with stakeholders	33	<i>Concordia can be used to validate features with the customer. () Strongly agree () Agree () Neither agree nor disagree () Disagree () Strongly disagree</i>	Level of agreement

Goal (to evaluate...)	#	Question	Metric
The capacity of detecting problems in the specification	34	<i>Concordia can detect incorrect declarations or other errors in the specification. () Strongly agree () Agree () Neither agree nor disagree () Disagree () Strongly disagree</i>	Level of agreement
The capacity of detecting differences between the SUT and its specifications	35	<i>The tests generated by Concordia can detect differences between the system under test and the requirement specifications () Strongly agree () Agree () Neither agree nor disagree () Disagree () Strongly disagree</i>	Level of agreement
In comparison with a framework	36	<i>Concordia is easier to use than a framework. () Strongly agree () Agree () Neither agree nor disagree () Disagree () Strongly disagree</i>	Level of agreement
Additional impressions about the solution.	37	<i>Please feel free to make additional comments.</i>	Observations (descriptive)

Table 25 – Semi-structured interviews rationale

Goal (collect qualitative data about...)	#	Question	Metric
Possible advantages of the solution	1	<i>What are the good points in using Concordia?</i>	Perception of the advantages
Possible disadvantages of the solution	2	<i>What are the bad points in using Concordia?</i>	Perception of the disadvantages
Possible improvements needed for the metalanguage	3	<i>Which improvements would you do in the language?</i>	List of improvements
Possible improvements needed for the tool	4	<i>Which improvements would you do in the tool?</i>	List of improvements
Possible improvements for the generated tests	5	<i>Which tests would you add to those generated by the tool?</i>	List of tests
Possible improvements for the solution	6	<i>Are there any other improvements that you think would be useful?</i>	List of improvements
Possible situations in which the solution may not be useful	7	<i>Are there any situations you would not use Concordia?</i>	List of situations

9.1.6. Controls and threats to validity

We employed the following controls for the questionnaire and interviews:

- Participants were instructed to inform if they had trouble understanding any question;
- Participants were asked to avoid communication during the activities;
- Participants were allocated far from each other for the duration of the interviews.

We highlight the following threats to the validity of the study:

- **Internal validity:** this threat concerns the effect that the treatment can cause in the outcome.
 - a) We used only one person to tabulate the data and to conduct interviews. Although precautions were taken to not influence participants along the process, it may have had some effect on them;
 - b) Training may have been insufficient, which could increase the difficulty to use the proposed metalanguage and the prototype tool We tried to mitigate this threat by offering support during the case studies and making the related documentation available;
 - c) Precision of collected data was reduced due to the impossibility of participants to collect more quantitative metrics;
 - d) Fatigue effects during interviews were mitigated by keeping the interviews short (10-15 minutes). We tried to reduce fatigue effects when participants answered the questionnaire by allowing them to fill it incrementally for three days.
- **External validity:** this threat concerns the generalization of observed results to a larger population, outside the sample instances used in the experiment. The study had a small sample size. We currently narrowed our approach to information systems, in order to limit the generalization of observations. Company selection also followed this criterion. Finally, the short duration of the study could not be mitigated due to the existing time restriction for the thesis.
- **Conclusion validity:** this threat concerns the relation between the treatment and the outcome. The study was possibly affected by random facts from environment (*e.g.*, company culture, participants' expertise, software types). To mitigate it, we adopted different instruments to collect quantitative and qualitative data (section 9.1.5). It was possible to intersect evidence emerged from practice, participants' opinions, researchers' observations, and literature, strengthening the study findings.

9.1.7. Analysis

The study used Technical Action Research (TAR). Wieringa (2014c) affirms that “*Technical action research is the use of an experimental artifact to help a client and to learn about its effects in practice. The artifact is experimental, which means that it is still under development and has not yet been transferred to the original problem context. A TAR study is a way to validate the artifact in the field. It is the last stage in the process of scaling up from the conditions of the laboratory to the unprotected conditions of practice.*”.

TAR has been used to validate methods in industry. For example, Morales-Trujillo *et al.* (2015) report that the combination of TAR and case studies was a successful experience to bridge the gap between academy and industry; Parra *et al.* (2017) used TAR to validate a model-driven method for gesture-based software interfaces; Morali & Wieringa (2010) used TAR to validate a method for specifying confidentiality requirements of outsourced systems.

Considering an approach, technique or model that has been designed and successfully tested with some artificial examples by the researcher, TAR can validate whether it (the approach, technique or model) can be used in real-world situations (WIERINGA, 2014c). In our case, TAR is used to validate if the proposed approach works with information systems in software development companies, considering our goal and research questions (section 9.1.2).

TAR consists of a five-step research cycle (WIERINGA, 2014c):

1. *Problem investigation*: determine what the unit of study is, what concepts are used to state the research questions about the unit of study, and what we already know about the research questions. **We defined our problem investigation strategy in section 9.1.1, the unit of study and the research questions in section 9.1.2;**
2. *Design*: consists of acquiring access to a client company, agreeing on an improvement goal for the client cycle, agreeing on what the researcher will do for the company and on how the researcher will collect data. **We provided details about how we contacted companies and participants**

in sections 9.1.3 and 9.1.4, and about the procedures and measurements in 9.1.5. All the participants were clarified about the case study before they start participating and, again, in kickstart meetings;

3. *Validation*: consists of assessing the risks of not being able to answer the research questions if the researcher executes the research design. **We detailed the controls and threats to validity in section 9.1.6;**
4. *Execution*: consists of the execution of the client cycle, part of which is the operationalization of the treatment plan (*i.e.*, to use the approach) already agreed on in the research design. Here, resources, people, time and places have to be agreed on to perform the tasks of the treatment. **The analyzed companies adopted the approach and used its prototype tool, and their feedback was collected every week during the studies. At the end of the studies, we formalized the results and impressions through a questionnaire and semi-structured interviews. We present and discuss the results in section 9.4;**
5. *Evaluation*: consists of analyzing the results. Observations are extracted from the raw data, possible explanations are searched for, research questions answered, and generalizations to other cases from the same problem class hypothesized. Limitations of these outcomes are stated explicitly, and the increment of knowledge achieved identified. **We evaluate the results in section 9.4.**

The entire TAR exercise is based on the assumption that what the researcher learns in a particular case (*i.e.*, a company) will provide lessons learned that will be usable in the next case (LEE & BASKERVILLE, 2003; SEDDON & SCHEEPERS, 2012, 2006; WIERINGA, 2014c). **During the case studies, we identified problems and made adjustments in the prototype tool based on companies and participants' feedback. We discuss these lessons learned in section 9.4.**

9.2.Quantitative data

We organized the qualitative data in the following subsections. Subsection 9.2.1 details companies' and participants' profiles. Subsection 9.2.2 considers the evaluated features' maturity and frequency of validation with stakeholders. Subsection 9.2.3 concerns with the understanding of Concordia specifications. Finally, the subsection 9.2.4 details the participants' perception about the approach and the prototype tool.

9.2.1.Profile

All the profile data consider the state-of-the-practice before using Concordia.

Figure 33 shows the prior experience of participants and the usage in companies regarding documentation artifacts and automated tests. Although participants also have prior experience with use cases and user stories, their great majority (89%) uses wiki or textual documentation in companies. Only a few participants (22%) had prior experience with automated functional or UI tests, and most (46%) do not use any kind of automated tests in their companies.

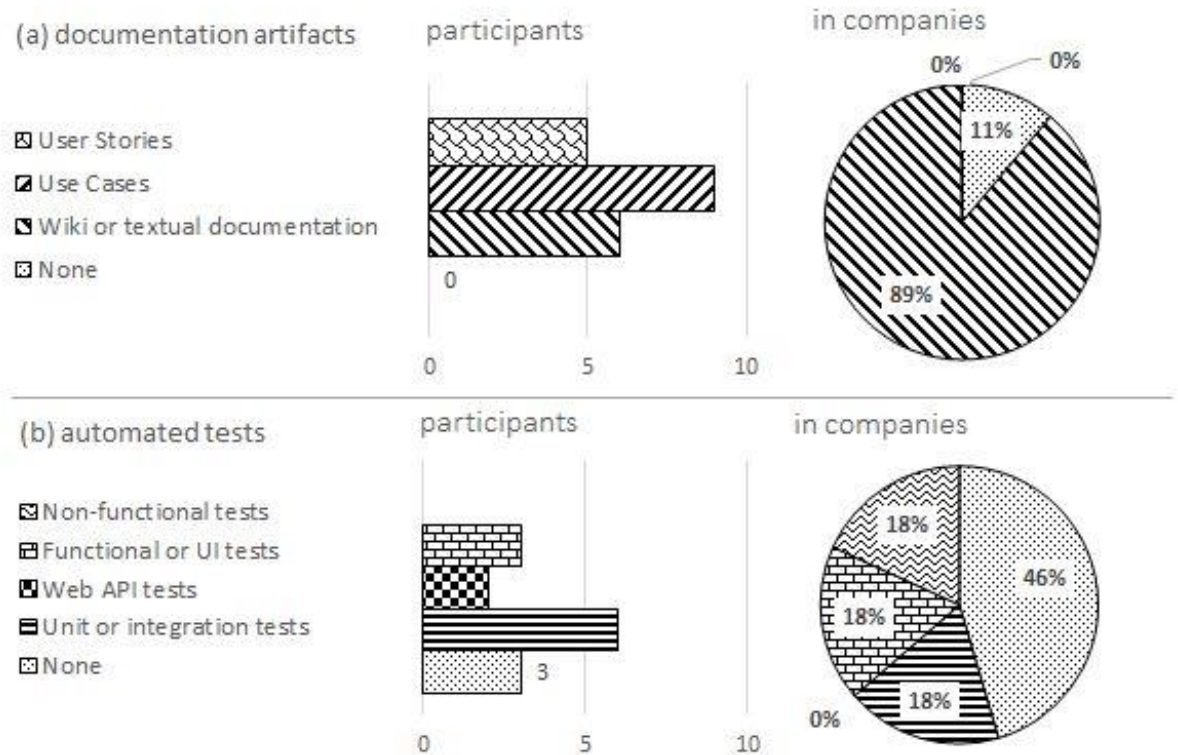


Figure 33 – Documentation artifacts and automated tests

Figure 34 shows the usage of requirements specifications for validation with stakeholders (a) and for producing tests (b). The validation with stakeholders occurred “occasionally” in most cases (67%). Since we also asked to participants about the frequency of validation with stakeholders (Figure 27d), we could ascertain that it occurred from 15 to 30 days on average. Requirements specifications were not used to produce any kind of test in most cases (46%).

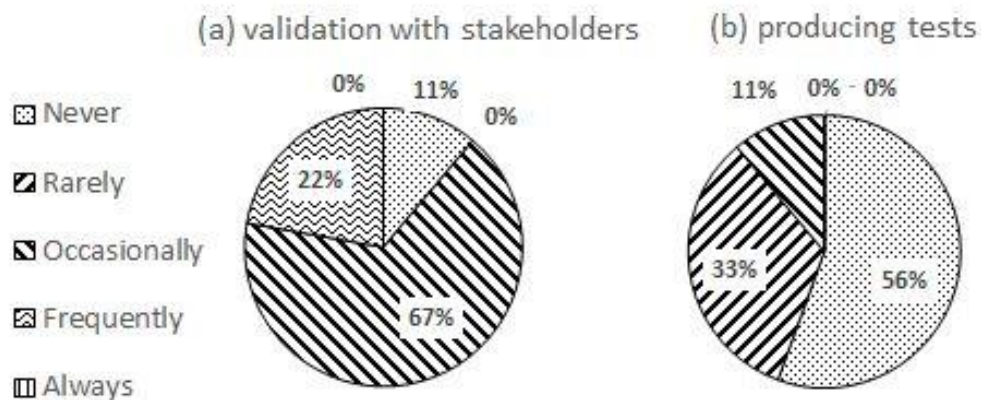


Figure 34 - Usage of requirement specifications

9.2.2.Features and validation

Figure 35a presents the average frequency of features specified with Concordia per company. The great majority of the companies (75%) specified from 5 to 9 features, while the other companies specified 10 features or more. (Figure 35b shows the average frequency in which these features that were validated with stakeholders by participants. Most participants (56%) validated up to 4 features. Figure 35c presents the maturity of involved features, *i.e.*, the elapsed time since they were deployed. The majority of the features (36%) has 7 to 12 months and only 9% can be considered mature (one to three years). Figure 35d shows the time interval in which features were validated with stakeholders. Most participants (67%) validated features with stakeholders in a range that varied from 15 to 30 days.

Figure

35

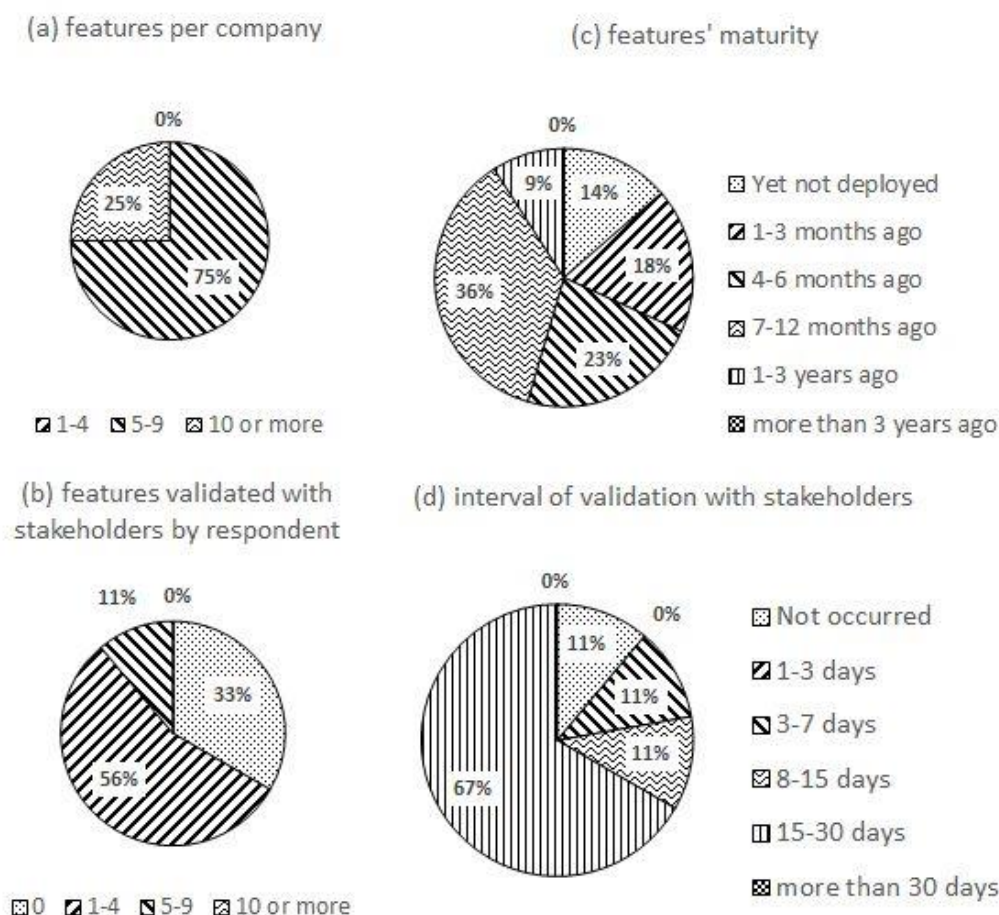


Figure 35 - Features, maturity, and validation

9.2.3. Concordia

Figure 36a presents the perception of participants about the difficulty of stakeholders in comprehending requirements specifications written with Concordia. Most participants (45%) affirmed that stakeholders could not understand some parts (medium difficulty), 33% affirmed that stakeholders could understand almost everything (small difficulty), and the other participants (22%) could not evaluate. We could verify from both the questionnaire and interviews that these parts were the related to two declarations: properties (constraints) of User Interface Elements and Databases. *These parts of the metalanguage were not intended to be validated by stakeholders but to allow the software team to express system's constraints or properties aiming to generate tests.* Participants did not report stakeholders' difficulties about any other parts of the language, such as Variants or Test Cases. *Therefore, the difficulty attributed to the results from Figure 36a are related to declarations that should not have being validated with stakeholders.* Furthermore, the questionnaire revealed that a hundred percent of the stakeholders did not receive any explanation about the metalanguage (Concordia) or the format of the requirements specifications before reading them.

Figure 36b shows the perception of participants about the difficulty of their coworkers to understand Concordia specifications. Most participants (56%) reported that coworkers could understand everything (none difficulty), 33% affirmed that they could not evaluate that, *i.e.*, they did not pay attention to that matter during the study, and 11% reported that their coworkers could understand almost everything (small difficulty). We also could verify that the most common difficulty was related to properties of User Interface Elements.

Figure 36c presents the initial perception of participants about their difficulty to understand Concordia specifications. Most participants (56%) reported that they could understand everything (none difficulty), and 44% reported that they could understand almost everything (small difficulty). During questionnaires or interviews, they reported that they could solve any difficulties by contacting the support or reading Concordia's documentation.

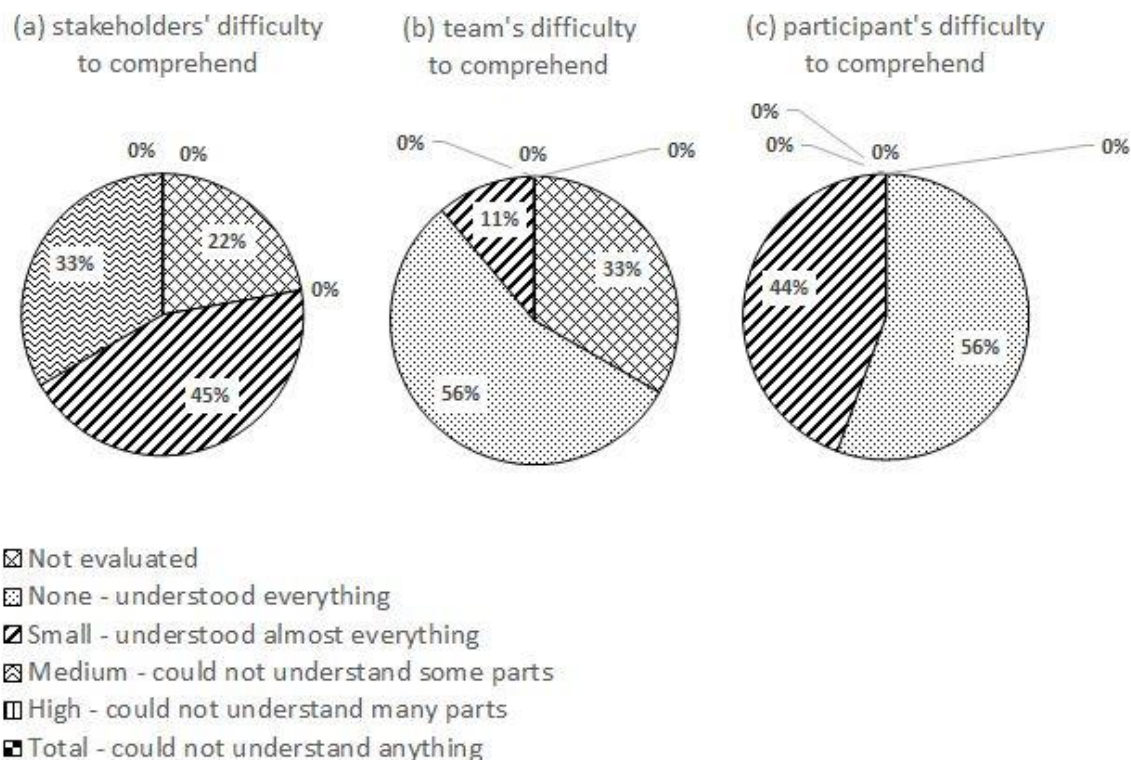


Figure 36 –Concordia’s reading comprehension

Figure 37 presents the perception of participants about writing requirements specification with Concordia (Figure 37a), and the amount of Concordia’s documentation that they affirm to have read before reading or writing them (Figure 37b). The great majority of participants (89%) reported that they had a small difficulty to write Concordia specifications. These difficulties were solved with the help of supporting tasks (*i.e.*, by contacting the researcher) or by reading the language’s documentation. Most of the participants (56%) affirmed that they read something of the Concordia’s documentation, and 44% affirmed that they read most of it.

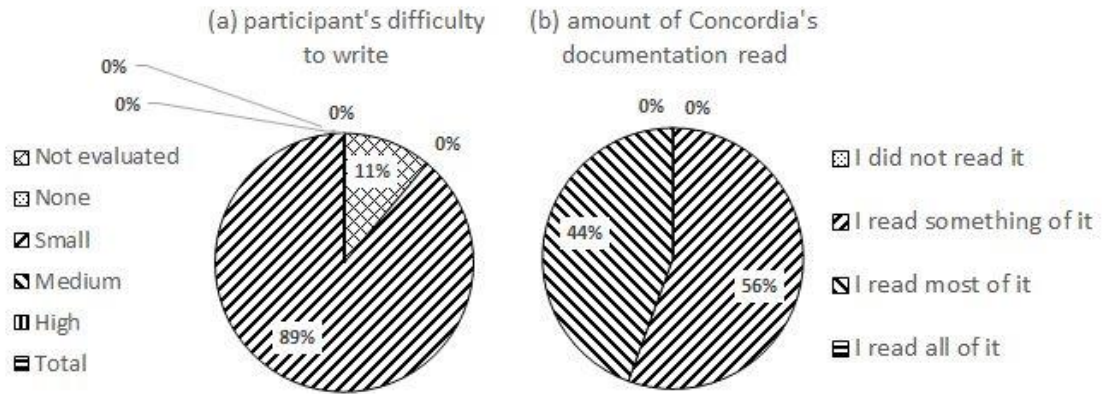


Figure 37 – Concordia's writing comprehension

Figure 38a presents the participants' perception of the time to write Concordia specifications. The majority (60%) reported that the time to write them was short. Figure 38b shows the participant's perception about the time to write Concordia specifications, compared to the time to write specifications using the document adopted by the company before trying Concordia (*e.g.*, wiki, use cases). The majority (56%) affirmed that it was faster to write specifications with Concordia.

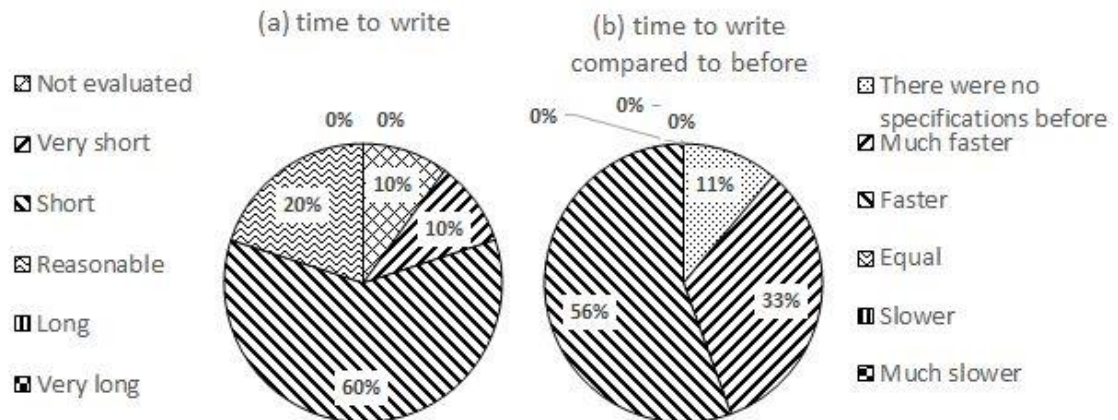


Figure 38 - Perceived time writing Concordia specifications

9.2.4. Prototype tool and approach

Figure 39 presents the participants' perception about the tests produced with Concordia in comparison with the tests produced before adopting it. Figure 31a considers the number of such tests. Most participants (67%) reported that the number of tests produced with Concordia was much larger than before using it.

Figure 31b concerns the easiness to produce such tests and the great majority of participants (89%) considered that it was easier to produce tests with Concordia. Figure 31c shows that 78% of participants considered the quality of the tests produced with Concordia much higher than they had before using it. Figure 31d shows that 67% of participants considered the time to produce tests with Concordia was much shorter than before using it.

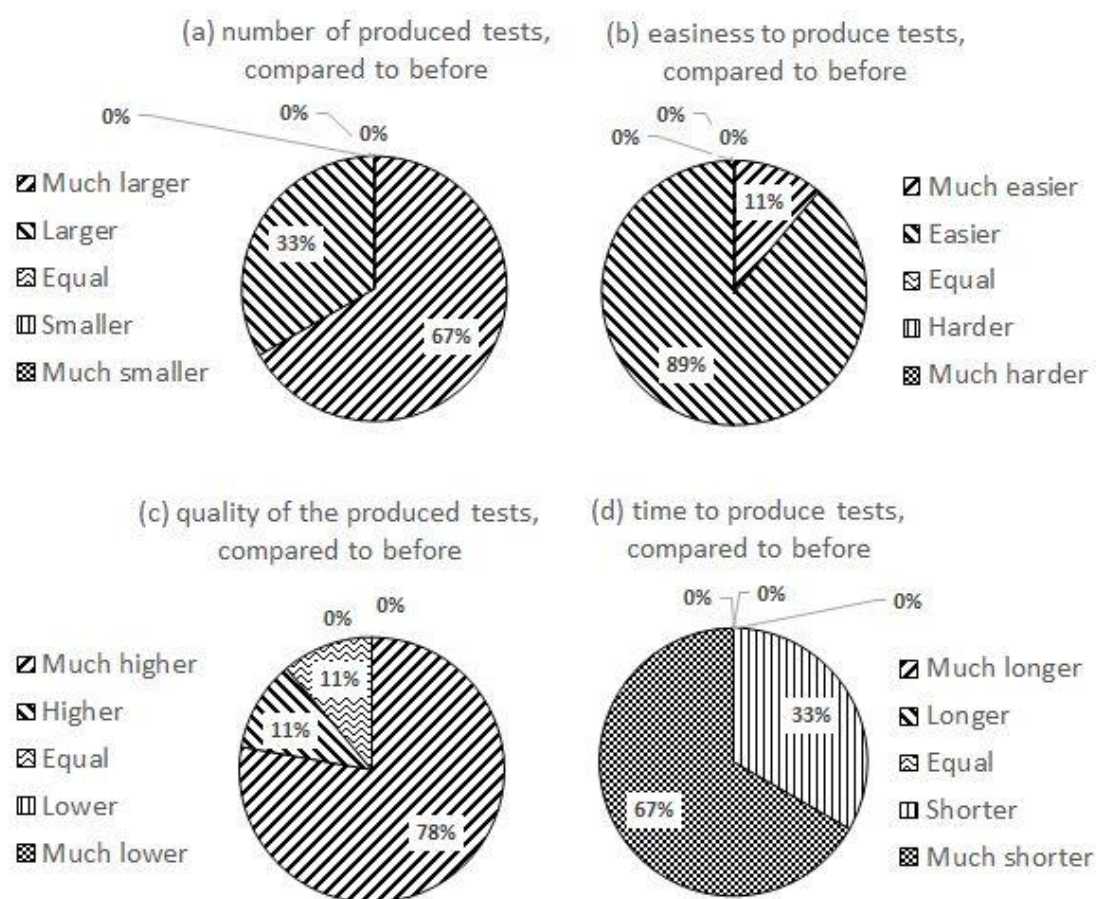


Figure 39 - Perception about Concordia tests

Table 26 shows the number of defects found per participant and company.

Table 26 - Defects found per participant and company

Participant	Defects	Company	Defects per company
<i>O-V</i>	3	O	6
<i>O-R</i>	0		
<i>O-A</i>	3		
<i>I-W</i>	4	I	5
<i>I-S</i>	1		
<i>H-W</i>	0	H	1
<i>H-V</i>	1		
<i>D-G</i>	0	D	1
<i>D-D</i>	1		
Total:	13	Avg:	3,25

Figure 40 presents the overall perception of participants about Concordia. Figure 40a reports that 89% of participants strongly agreed that Concordia can be used for validation with stakeholders. Figure 40b shows that 78% of participants strongly agreed that using Concordia is easier than developing tests with a framework. Figure 40c reports that 78% of participants strongly agreed that Concordia can detect errors in specification documents. Finally, Figure 40d shows that 78% of participants strongly agreed that Concordia can detect differences between the system and the specifications.

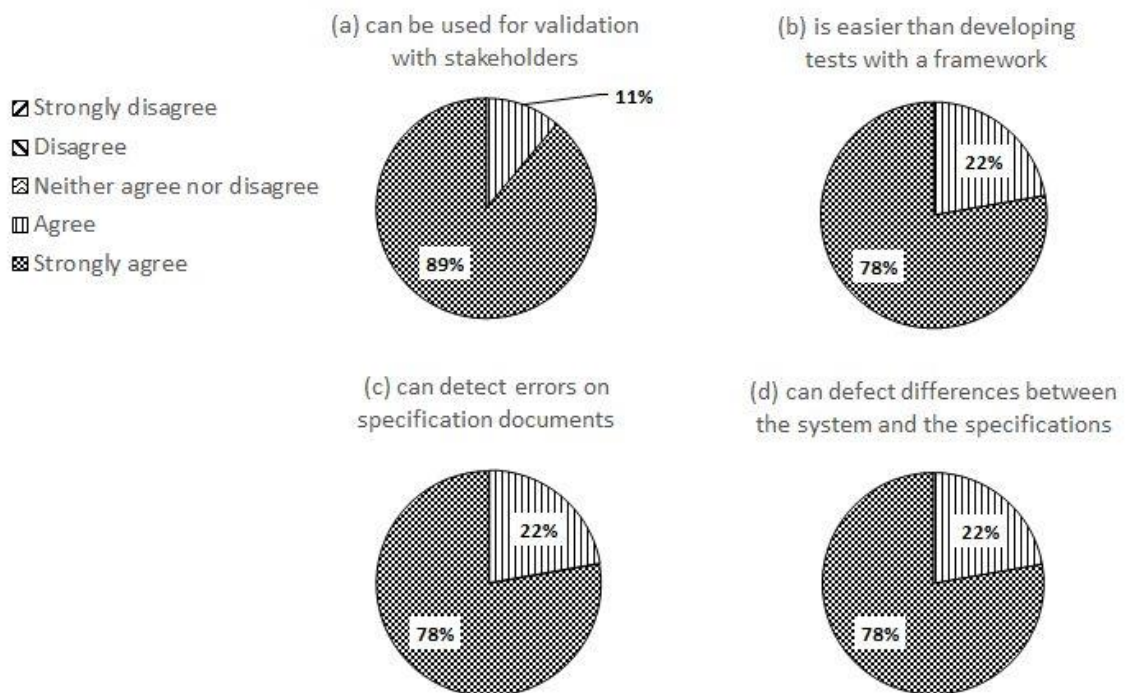


Figure 40 - Overall perceptions on Concordia

9.3. Qualitative data

We collected most qualitative data from questionnaires and semi-structured interviews, via smartphone or Skype⁷¹. Details are presented below.

9.3.1. Interviews

Interviewed participants were the same that answered the questionnaire. They reported their impressions in an informal way, making oral comments about the initial questions and any other subjects.

About the question “*What are the good points in using Concordia?*”, most participants reported: (a) discussing requirements with customers; (b) writing requirements as a way of getting the tests done; (c) it is not complicated. Three participants (O-A, O-V, I-W) reported that the language is easy to write: *e.g.*, “*it (Concordia) is intuitive, I don’t need to consult the documentation to write things (...)*” (O-V).

⁷¹ <https://www.skype.com>

About the question “*What are the bad points in using Concordia?*”, most participants reported “*none*” or “*none so far*”. Two participants (O-V, D-G) reported the needed to find the identification of widgets in the user interface: “*...there are cases in which the number of elements on the screen is large*”. However, when asked about whether they had used the browser plug-in created to capture these identifications automatically via Record and Playback (Katalon Concordia, see chapter 8), both reported that it helped them to mitigate the problem: “*yeah, after using the plugin, the problem was solved.*” (D-G); “*yes, I had just one problem with a (third-party) multi-select component that I needed to inspect manually to find the identification. The plugin resolved the other cases.*” (O-V). One participant (D-D) reported that “*complex cases are harder to specify and the documentation could have more examples (...)*”.

About the question “*Which improvements would you do in the language?*”, most participants reported “*none*” or “*none so far*”. Three participants (O-A, O-V, I-S) made positive comments after reporting “*none*”: “*from the customer point of view, the language is very easy to understand*” (O-A); “*I really liked the states, the fact that they avoid me to rewrite things*” (O-V); “*The language is easy. I wrote documentation for two projects: one in English and the other one in Portuguese. For the project in Portuguese, I just tried to write the things in the same way and it worked.*” (I-S). One participant (O-A) also indicated that the constraint for required UI Elements (see UI Element) could be written as “*required*” instead of “*required is true*”, because a customer asked about the “*is true*” part.

About the question “*Which improvements would you do in the tool?*”, all participants reported “*none*”. Four participants (H-W, O-V, I-S, I-W) made additional compliments, e.g., “*I felt it is very complete for a prototype*” (I-S).

About the question “*What do you think of the tests generated by the tool?*”, all participants reported positive impressions. For example, I-W reported “*Better than I expected*”; I-S reported “*They cover very well the important cases*”; H-W reported “*Very good, very complete*”. One participant (O-V) also reported: “*I also liked the fact that I can write my own tests if I need to*”.

About the question “*Which tests would you add to those generated by the tool?*”, all the participants reported no tests.

About the question “*Are there any other improvements that you think would be useful?*”, most participants answered “No” and some of these gave some complement, e.g., participant D-G reported “*No, it already takes a screenshot when the tests fail and generates a test report.*”. Participant O-V reported “*I had a complex user interface component whose interaction took a series of steps. I would be interesting if I could encapsulate these steps in that component instead of letting them in the Variant. In this way, a single step that interacts with it (in the Variant) would be replaced by the steps that I had encapsulated in the component. (...)*”.

About the question “*Are there any situations you would not use Concordia?*”, most participants reported answers like “*I think there aren't*”. Participant O-V reported “*Maybe in a system with very strange rules or rules very difficult to specify, like a game*”.

9.3.2. Questionnaire

The questionnaire included open-ended questions to collect qualitative data.

About the question “*Whether you validated requirements with customers, please inform about any difficulties*”, most participants left it blank. Three participants (O-A, D-G, I-W) reported that they were asked about properties of UI Elements (see 6.1.10), e.g., “*UI rules, such as "required is true"*”(O-A).

About the question “*Whether you had any difficulties to understand Concordia specifications, please inform them*”, all participants left it blank.

About the question “*Whether you had any difficulties when writing Concordia specification, please inform them*”, most participants left it blank and the other ones gave generic answers such as “*In the beginning, when I was learning it (Concordia), I needed to resort to the documentation*” (I-W).

About the question “*What do you think takes longer when specifying requirements with Concordia? Why?*”, all the participants agreed about being the time taken to find the identification of the user interface elements, e.g., “*In my case, it was the time to find the elements on the page (...)*” (O-A).

About the question “*Did you discover any other problems in your application while using Concordia? Which ones?*”, four reported problems with the user interface, e.g., “*Yes. (...) input fields that should have a fixed width were growing unexpectedly when they received long input data*” (O-A). One participant (H-W) reported four relevant problems: “*Yes, I discovered problems related to business rules, a problem with our testing environment, and it became clear to me that we have some usability problems as well as the low maintainability of the application*”. Later we asked the participant (by phone) for details about this specific answer, and it was informed that: (1) most business rules problems were related to differences between implemented constraints and desired constraints, that is, constraints implemented for some features of the application were behaving differently from what they expected. The main reason was that the old documentation had no details about them. Thus, probably the programmer adopted the constraints he/she judged more appropriate for the moment; (2) the testing environment was getting complex to configure and the team was wasting too much time to replicate the execution environment. Concordia helped to make clear to management that they had to invest in improving the testing environment; (3) Usability problems were detected when executing Concordia tests in different browsers and screen sizes. Some user interface components were not rendering the way they expected (they were overlapping or changed their position strangely); 4) The maintainability problem was related to the lack of naming patterns for user interface components in some cases.

About the question “*Please give us additional comments if you have some*”, most participants left it blank and one participant (D-D) reported “*It would be nice to have even more examples on the documentation.*”.

9.3.3. Support tasks

We collected some qualitative data from support tasks, through messages or during *in-loco* visits. Many of these data are related to common doubts about the language. The three most common doubts about the language were:

- 1) About how to write a particular sentence in a Variant, *e.g.*, participant H-W asked via e-mail “*How can I (write a sentence to) select the option “legal entity” (...)”*”;
- 2) About how to declare UI Element properties, *e.g.*, participant D-D asked via Skype: “*How can I define the maximum number of characters (accepted by a UI Element)?”*”;
- 3) About how to declare or use States, *e.g.*, participant D-G asked via e-mail: “*Is it possible to create many features that produce the same state?”*”.

We also collected feedback about eventual problems with the prototype tool. Five defects were identified by participants or with their help. Table 27 describes them. Three of them (#1, #2, and #5) were due to simple coding mistakes or changes in third-party libraries (#2), one (#3) was due to a mistake in how the algorithm was implemented (not in the algorithm itself), and one (#4) was a problem in the adopted algorithm. This latter was relevant to the approach since it influences the generated test cases in specific cases. To summarize, a UI Element can have a property “format” declared with a regular expression that indicates the format accepted as valid. In this case, our approach uses a finite-state automaton to produce two test data values: a value that does not match the specified regular expression, *i.e.*, a value whose format is considered invalid, and a value that does match the regular expression (*i.e.*, format considered valid). However, when another UI Element property that constraints the accepted values in some way is defined – *e.g.*, a property like “value is in [“Bob”, “Alice”]” would constraint the values accepted as valid to “Bob” and “Alice”, and a property like “maximum length is 10” would constraint the values’ length to 10 –, these constraints cannot be satisfied together, due to a current limitation in the constraint solver (the finite automaton also does not accept any other constraints). Thus, we removed the test case that explores an invalid format when another value constraint is used.

Table 27 - Defects found in the prototype tool with the help of participants

#	Description	Issue ID ⁷²
1	Parameter --language was being ignored.	9
2	Short flag aliases of the command line interface were not working, <i>e.g.</i> , to use -d instead of --directory.	13
3	Problem combining more than one precondition state.	20
4	When a UI Element had value or length constraints, a test case that explores an invalid format should not be generated, due to the lack of a more general constraint solver.	14
5	Spaces in files names were being replaced with %20	23

Another relevant feedback collected during support tasks was the need for specific actions in Variant sentences (section 6.1.11). Instead of trying to define all the possible actions and variations beforehand, we defined a basic set and incremented it on demand, based on users' needs.

9.4. Discussion

Endres & Rombach (2003) point out that to validate an artifact in a real context is the principal means to obtain knowledge in Software Engineering. Using our approach in software development companies allowed us to gain experience and generate knowledge through the perceived effects and lessons learned.

9.4.1. Lessons learned from the client cycle

The iterative approach of TAR helped us to reason about the incremental feedback received from companies and participants during the case studies. We used this feedback for:

1. Validating the adequacy of the language vocabulary;
2. Augmenting the number of supported actions in Variants and Test Cases;

⁷² Issues IDs as they were registered at the project's repository, available at: <https://github.com/thiagodp/concordialang/issues?q=is%3Aissue+is%3Aclosed>.

3. Fixing defects;
4. Validating the adequacy of generated test cases, including test data and oracles;
5. Validating the correctness of generated test scripts;
6. Evaluating the usability of the tool, *e.g.*, whether users had difficulties to understand and use its parameters;
7. Improving the tool's documentation.

We also had the opportunity to develop the three roles identified by Wieringa (2014c): Designer, Helper, and Researcher. As designers, we created artifacts whose objectives were to resolve problems present in industry. These artifacts were inserted into companies during the engineering cycle, and we acted as helpers to apply the proposed treatment (*i.e.*, approach, language, and supporting tool) and assess its functioning. As researchers, we gather feedback to learn, to improve the artifacts, and to analyze the resulting effects. This analysis enables additional adjustments and improvements to the artifacts.

At the end of the studies, *organizations achieved the stated objectives, i.e.*, they increased the number and quality of their test cases, and gained an artifact that can be adopted incrementally and used to both discussing requirements with customers and checking whether their software corresponds to the specified functional requirements. Their employees also benefit from the case studies since they received some training in Agile DSLs, test automation, and learned a new metalanguage, Concordia.

An increase in the number of tests, especially in regression tests, provides a safety net (BECK, 2003) that allow developers to be more confident about maintaining legacy systems or changing new systems. Companies that participated in the case studies identified during informal conversations that changes that produce “ripple effects” in other features are recurrent and often cause problems due to the lack of regression tests – notably in legacy systems. Since we observed that using Concordia has motivated them to invest in writing specifications because they noted all the collateral benefits, we expect that they can mitigate such problems and keep

improving the quality of their systems, including the legacy ones. Rosa (2011) indicates that using tests to support maintenance can reduce the number of introduced defects and improve maintainability.

All the companies in the case studies affirmed that they shall continue using Concordia and will adopt it for new systems. Some of them also showed interest in adopting it for older systems, incrementally, in the short to medium term.

9.4.2. Research questions

Regarding **RQ1** (*How difficult is it for stakeholders to understand requirements specified with Concordia?*), Figure 27 shows that stakeholders had some difficulty to understand Concordia specifications: 33% could understand almost everything and 45% could not understand some parts. However, as we pointed out in section 9.2.3, we could determine – through the questionnaire and interviews – that those difficulties were *only related to parts of the metalanguage that were not intended to be validated by stakeholders, i.e., UI Elements' properties and Databases' properties, or not intended to be validated without the support of the team*. The questionnaire also revealed that a hundred percent of the stakeholders did not receive any explanation about the metalanguage (Concordia) or the format of the requirements specifications before reading them.. Considering these data, and mainly considering that stakeholders *did not report any difficulties with other parts of the metalanguage*, we could conclude that **stakeholders had few difficulties with the parts addressed to them** (*i.e., Features, Scenarios, Variants, and Test Cases*). *This comprehension is important because it details the results from Figure 27 and clarifies our conclusions about the overall context – and, thus, about RQ1.*

Regarding **RQ2** (*Is the metalanguage useful to validate requirements with stakeholders?*), when participants were asked about whether they think that Concordia can be used for validating requirements with stakeholders (Figure 40a), the great majority (89%) rated as *strongly agree* and the others (11%) as *agree*. Considering these data and also considering that the metalanguage is based on Agile DSLs that has been used in industry (CURCIO et al., 2018; INAYAT et al., 2015; SCHÖN;

THOMASCHEWSKI & ESCALONA, 2017) to validate requirements with stakeholders (ADZIC, 2009, 2011; SMART, 2014; WYNNE & HELLESØY, 2012), we answer it positively, that is, we claim that *Concordia is useful to validate requirements with stakeholders*. Besides, Features and Scenarios in Concordia can be written in natural language without any computing jargon. Even if a software team does not succeed in validating Variants or Test Cases (whose sentences describe expected interactions with the user interface) or even UI Elements (that define constraints related to business rules), the metalanguage could be used partially to get successful validations.

Regarding **RQ3** (*How difficult is it to specify requirements with Concordia?*), the great majority of participants (89%) evaluated the *difficulty to write* specifications in Concordia as *small*. When asked about the *amount of documentation* about Concordia that they had read (Figure 37b), 56% answered *something of it*, while the other participants (44%) answered *most of it*. Qualitative data showed that doubts were clarified with the help of the documentation – as is normal and expected in any new language, tool, or approach. Common doubts clarified through support tasks are identified in section 9.3.3. Most of them could have been clarified using the language documentation. Therefore, we claim that *Concordia has a small to medium-difficulty for writing specifications*. Most of that difficulty – as we pointed out in 9.3.3 – was about the vocabulary for writing Variant sentences, the properties of User Interface Elements, and the use of States. All of them could be solved by reading the documentation or contacting the support.

Regarding **RQ4** (*How time-consuming is it to specify requirements with Concordia?*), the majority (60%) considered the time to write Concordia specifications as *short*, in comparison to what they used to do before (Figure 38b). None of them considered it *long* or *very long*. Qualitative data also indicates that most of the time invested in writing specifications is consumed by the identification of user interface elements, which often occur for systems that were already implemented. We also identified that participants could decrease this time by using a complementary tool, created by us (Katalon-Concordia, see chapter 8), although it is only available for web applications. Furthermore, the prototype tool can help software teams that adopt naming patterns (*e.g.*, camel case, pascal case, snake case, kebab case) for

identifying user interface elements. Thus, specifying requirements in Concordia is *usually fast*, especially for applications that have at least one of these characteristics: (a) follow a naming pattern; (b) have not yet defined identifications (*i.e.*, specifications are written before the features are implemented); or (c) can benefit from the complementary tool.

Regarding **RQ5** (*Is the approach useful to verify the compliance of a system with its requirements?*), when asked about whether Concordia can detect differences between their system and its requirements (Figure 40d), the great majority of participants (78%) answered *strongly agree* and the others (22%) answered *agree*. Furthermore, the tests produced by Concordia are sensitive to changes in requirements (both actions, test data, and oracles) – as demonstrated in section 8.3. Therefore, we can answer it positively.

Regarding **RQ6** (*How complete are the tests generated with the approach?*), the great majority of participants (78%) considered the *quality* of the tests generated by Concordia as *much higher* than before using it (see Figure 39c). Moreover, compared to before, the *time* to produce them (Figure 39d) was considered *much shorter* (67%) or *shorter* (33%), and the *easiness* to produce them (Figure 39d) was considered *easier* (89%) or *much easier* (11%).

Finally, regarding **RQ7** (*Is the approach useful to discover defects?*), Concordia detected a total of 13 defects in the four companies' systems, during the case studies (see Table 26). The average maturity of evaluated features is *low* – only 9% has 1-3 years, and none has more than 3 years, see Figure 35c. However, it is *less probable* that Concordia can detect defects in *mature features* since they were supposedly more tested and used in production for a longer time. Nevertheless, it was useful for detecting defects and, in some cases, it was also useful for detecting other problems, such as user interface problems and maintainability problems (section 9.3.2).

9.4.3. Conclusions of the study

The object of study in engineering sciences is an artifact in a context of use (WIERINGA, 2014a; WIERINGA; DANEVA & CONDORI-FERNANDEZ,

2011). Engineering researchers iterate between (re)designing artifacts for use in a class of contexts and investigating artifacts that interact with contexts of this class. In this strategy, researchers start their investigations under ideal conditions in the lab and finish them under realistic conditions in the field. During the process, artifacts are scaled up to practice, and generalizations are increasingly targeted at field conditions – a process referred to as *lab-to-field generalizations* (WIERINGA & DANEVA, 2015). Lab-to-field generalization is a form of technology validation (R. GLASS; VESSEY & RAMESH, 2001; ZELKOWITZ & WALLACE, 1997). Research methods that can be used in technology validation include simulation, technical action research, and statistical difference-making experiments in the lab or in the field (WIERINGA, 2014b).

In our case studies, we used TAR to validate the artifacts in field conditions (see 9.1.7). Although we could not apply more measures that could strengthen our findings – due to companies’ shortage of time to adopt more controls –, the data and feedback received during our studies and collected at their end, give us enough evidences to conclude that the artifacts can be used successfully in field conditions, in similar environments, for achieving the desired outcomes. More specifically, we can use analytical induction (WIERINGA & DANEVA, 2015; ZNANIECKI, 1968) to confirm that an explanation constructed for one case study is also valid for other cases studies with similar architecture, but also differ from each other (ROBINSON, 1951; TACQ, 2007; YIN, 2003).

Studied companies have a similar architecture (*i.e.*, environment) in relation to they be *small or micro software companies* that develop *information systems*. Differences are related to the type of (information) systems produced (see Table 22), the size and heterogeneity of their teams, and the adopted technologies (*e.g.*, frameworks, programming languages, tools). In this context, they had very similar results (9.2 and 9.3), and our research questions could be answered in the same way considering all of them (9.4.2).

Therefore, we argue that for small or micro software companies that develop information systems:

1. Requirements specifications written in Concordia can be used to validate features with stakeholders;
2. Concordia specifications are considered easy to understand;
3. Concordia specifications are considered having medium easiness to write;
4. Static verification is capable of detecting wrong declarations and other problems in the specification;
5. The tests generated from Concordia specifications can detect differences between these specifications and the systems under test;
6. The tests generated from Concordia specifications can detect defects in features with low maturity.

Although we expect that medium-sized software companies may also have successful results, we could not evaluate that yet. The same holds true for other types of software.

9.5. Concluding remarks

This chapter detailed the evaluation of our approach through case studies with software companies. During the study, these companies specified requirements using the Concordia metalanguage, validated the requirements with stakeholders, and used a prototype tool for statically checking the specification and generating functional tests. Generated functional tests aimed at both verifying the compliance of companies' systems with the specification and discovering defects. Companies were accompanied during the studies and feedback was collected incrementally and used to improve the studied artifacts (approach, language, tool). Technical action research was used during the process. At the end of the studies, data were collected through questionnaire and semi-structured interviews. All the data were analyzed, research questions answered, lessons learned were presented, and their effects and conclusions were weighted. In summary, the approach presented in this thesis had positive results in the analyzed companies, regarding both validation and verification activities.

10 Epilogue

The battle of getting better is never ending.

- Antonio Brown (NFL player)

This chapter presents the conclusions of this thesis.

10.1. Conclusions

The following sub-sections consider the research questions and contributions of this thesis.

10.1.1. Research questions

Revisiting the main question of this work, “*Can Agile DSLs combined with our approach serve for both validating and automatically verifying applications effectively?*”, we can argue that:

- **About their use for automatic verification:** Our approach adopted a (flexible and adaptable) restricted natural language and used a lexer, a parser, and Intent Recognition to understand its sentences – written with Agile DSLs. The approach could use a series of techniques and algorithms to produce state-based test scenarios and relevant test data and oracles, also considering traceability and reduction concerns. The produced test scripts were able to reveal defects in studied companies’ applications (13 defects in 4 companies). No manual intervention was used to complete or change these test scripts. Some companies also found that these test scripts could help them to detect usability and maintenance problems in their applications. Furthermore, the approach could reveal syntactical, semantic, or logical errors in requirements specified by the studied companies. Therefore, we consider that the approach was

effective for automatically verifying requirements and applications. Future research can investigate its use with more mature applications and other types of applications.

- **About their use for validation:** Agile DSLs are already used for validation in most companies that adopt agile requirements engineering practices (CURCIO et al., 2018; INAYAT et al., 2015; SCHÖN; THOMASCHEWSKI & ESCALONA, 2017). Book authors (ADZIC, 2009, 2011; GÄRTNER, 2012; GREGORY & CRISPIN, 2010; SMART, 2014) also have been shown their efficacy in discussing and validating requirements with stakeholders. During our multi-case study with software companies, few of them could validate Concordia specifications – that use Agile DSLs – with stakeholders. Although software team members did not provide prior explanations about Concordia to stakeholders (customers), they affirmed having received positive feedback about the comprehension of Concordia specifications – *i.e.*, customers considered them easy to understand. Due to time restrictions and the impossibility to participate in the projects more intensively, the evaluation of validation aspects did not consider usage scenarios in the form of Test Cases. Variants served as usage examples and their validation with stakeholders got very positive results. Therefore, we can affirm that Concordia can be effectively used for validating requirements. We cannot yet affirm whether the produced usage scenarios – in the form of Test Cases – help (or not) with this validation.

Revisiting the secondary research questions:

- *How can Agile DSLs be used for generating full-featured test scripts?*
Briefly, we: (a) combined a restricted natural language, a lexer, a parser, and NLP techniques to recognize declarations; (b) used a state-based approach to generate test scenarios; (c) mixed classical testing techniques (as data test cases), user interface properties' constraints, a SQL-like query language, external data sources, and a constraint solver, to produce test input data and test oracles; (d) mixed test scenarios, test input data, and test oracles to produce test cases and used NLP to transforming them into natural language declarations; (e) transformed test cases into abstract

test scripts and, finally, used plug-ins to transform abstract test scripts into test scripts (source code). We detail the approach in chapter 7.

- *Can test scripts generated from Agile DSLs reveal defects in existing applications?*

Yes, the generated test scripts revealed 13 defects during the case studies (in 4 companies). Involved applications were already tested using traditional approaches, which shows the efficacy of the produced test scripts. The mentioned defects were discovered using the default configurations, that adopts minimization strategies. These minimization strategies reduce the number of generated test cases (aiming to reduce test time) and, therefore, also reduce test coverage. This coverage is reached over time – the more tests a company generates and executes, the more defects can be detected. Algorithms use new random seeds to pick different paths on each execution. Defects can be detected earlier by avoiding minimization strategies. However, test time increases substantially.

- *Can an approach for V&V that uses Agile DSLs reduce test time and costs?*

Yes, test time and costs are reduced by at least four components: (i) static error checking can detect problems before the tests start (*e.g.*, vague or erroneous declarations in Variants, conflicts between constraints) (ii) test scripts are produced much faster than by the equivalent manual approach; (iii) coverage is usually greater than the manual approach (*i.e.*, developers frequently would not remember of all the test cases that the approach generates) in less time (*i.e.*, developers frequently would not have time to program all the test cases that the approach generates); (iv) produced test scripts become regression tests and can detect defects introduced by maintenance tasks.

- *Can an approach for V&V that uses Agile DSLs be used for preventing defects?*

Yes. Informal reviews and collaborative work can detect imprecisions, incompleteness, and ambiguity in requirements, especially in business-related declarations, *i.e.*, Features and Scenarios. Static error checking can detect problems in technological-related declarations, *e.g.*, vague or erroneous declarations in Variants, conflicts between constraints in UI

Elements. Validation with stakeholders can detect confusing or inappropriate requirements, in Features, Scenarios, Variants, UI Element properties, and Test Cases. Since changes in the specification make the approach to produce new test cases, their impact in the application can be evaluated prior to its maintenance, *e.g.*, the software team can know which parts of the application do not pass the tests anymore. Hence, the approach allows test-driven maintenance. Regression tests can also detect problems introduced by maintenance tasks before a version is released to customers.

10.1.2. Contributions

The main implications of this work are:

1. a new metalanguage for writing agile requirement specifications that can be used for both V&V activities;
2. the first approach to generate *full-featured* ready to use test cases and test scripts from agile requirements specifications;
3. the first *integrated* approach for V&V of agile requirements specifications;
4. the assessment in industrial context of the proposed approach.
5. new techniques for producing test scenarios, test data, and test oracles based on agile requirement specifications;
6. integration of state-of-the-art techniques for minimization, selection, and prioritization of requirements, test cases, and test scripts;
7. an open source prototype tool that implements most of the proposed approach and can support its adoption by companies.

Additional contributions include:

- a) A comparison of metalanguages for agile requirements specifications;
- b) A comparison of solutions for natural language processing;
- c) A mini-process and its maintenance recommendations to increase the chances of adopting the approach successfully;

The proposed metalanguage, Concordia, has the following possible uses:

1. Specifying requirements in more than one spoken language and using plain-text format;
2. Validating requirements with stakeholders;
3. Discussing requirements and test cases among the software team (use as communication media);
4. Specifying functional test cases in (restricted) natural language;
5. Checking requirements specifications for syntactical, semantic, and logic errors;
6. Generating, executing, and analyzing *full-featured* functional test cases and test scripts;
7. Using external data sources, such as databases, for creating constraints about user interface elements and producing test cases;
8. Discovering defects, especially in recent applications;
9. Verifying the compliance of an application with its Concordia specifications;
10. Supporting Behavior-Driven Development, Acceptance Test-Driven Development, and Specification by Example;
11. Supporting the adoption of functional tests in novel or legacy applications;
12. Supporting test-driven maintenance;
13. Using a requirements-first approach for test-driven maintenance – that is: change requirements, use the tool to produce the respective tests, and then modify the application to pass these tests;
14. Separating business declarations from test-level declarations;
15. Defining test case events in (restricted) natural language, for configuring the state of applications before or after the test scripts run;

10.2.Future work

The following sub-sections describe ideas and possibilities for research and other improvements.

10.2.1.Approach

Future research directions may consider:

1. *Conducting broader studies* with software companies:
 - a) Accompanying software projects from requirements elicitation to the release of versions, aiming to *observe the effects of the proposed approach* with respect to the validation with stakeholders, the communication among the team (with different company sizes), the development and maintenance of features (*e.g.*, test-driven development with Concordia, test-driven maintenance with Concordia), final tests before releases, etc.;
 - b) Investigating how the approach performs with *other types of software* (*e.g.*, text editors, spreadsheet software, presentation editors, diagram editors, e-mail programs, database designers, multimedia software, simulation software); and
 - c) Investigating whether there are *significant differences between software platforms* in relation to the vocabulary of Variants and Test Cases for validation with stakeholders. We could only verify the approach with web applications. Currently, the approach already supports the generation of test scripts for web applications, mobile applications (both native and web-based), and desktop applications;
2. *Comparing and improving involved approaches and techniques*:
 - a) *Comparing combination approaches, selection approaches, and prioritization approaches* – to evaluate which approach (of each group) has the best effectiveness, *i.e.*, which balances better defect detection (coverage) and execution time;
 - b) *Comparing approaches for Intent Recognition* – in terms of *precision* and *recall*, aiming to evaluate which one has better results with Concordia;
 - c) *Improving the constraint solver* or adopting a new constraint solver, to expand the flexibility of supported constraints, aiming to detect new types of defects in test cases;

- d) *Investigating an additional approach for the oracle generator*: to use NLP to negate Variant oracles in specific cases. Currently, when the approach selects a data test case that produces an invalid input value and the corresponding behavior for handling that invalid input is not specified (in an Otherwise sentence), the generated test case is flagged with “@fail” to indicate that its oracle should fail. A more precise technique is probably negating the original oracles using natural language processing, to ensure that their original expectations do not happen;
 - e) *Analyzing the impact of changes in Concordia specifications* to determine all the scenarios necessary to be retested. We believe that since our approach uses State-based dependencies between Variants, when a Variant is changed, we can determine all the affected Variants. Consequently, we can determine the Test Cases to update (generate again), to transform into test scripts, and to run. This can reduce significantly the time to execute regression tests without losing effectiveness. Another important information is about the Import declarations. Since we can use it to determine which files are affected by a particular file, we can also reduce retest time when declarations other than Variant are changed. In both situations (States and Imports), it may be necessary to use a *version control system* to establish what was changed in a specification file (*i.e.*, retrieving the changed lines and columns to determine the changed declarations);
 - f) *Creating an approach for symbolic execution* of Concordia declarations (which includes a constraint solver) to improve its static verification;
3. *Comparing the effectiveness of the produced test cases* to those produced from other requirements specification documents, such as Use Cases;
 4. *Investigating whether the adopted approach and techniques can be used with other requirement specification documents*, such as Use Cases. This may include NLP techniques, test case generation approaches, etc.;

5. *Generating user interface prototypes from Concordia declarations.* Concordia specifications declare interactions with user interfaces and involved UI elements. These declarations can probably be transformed into low-fidelity UI prototypes such as wireframes or mockups.

10.2.2.Metalanguage

Future research about the metalanguage may include:

- *Investigate whether the keyword-based approach can help to improve Concordia in some way.* For example, a participant of the multi-case study indicated that it would be interesting whether Concordia could allow declaring a high-level sentence that had corresponding low-level sentences declared somewhere. The participant affirmed that such declaration could allow replacing a small group of sentences whose only purpose is to interact with a complex user interface component with a single sentence that abstracts the interaction. This kind of declaration is common in the keyword-based approach;
- *Improving the syntax and vocabulary* aiming to increase language flexibility, supported actions, and its capacity to express constraints and expectations;
- *Performing a full syntax comparison with other metalanguages*, such as those presented in chapter 4. Such comparison can help organizations or individual developers to know the metalanguages and decide which to choose. It may also give researchers insights about the most helpful language declarations, in order to improve existing languages.

10.2.3.Tools

Possible improvements for the prototype tool are:

- a) *Generating graphs of relationships*, such as dependency or usage among features, scenarios, states, or files. This feature can provide better visualization about the specification and help to identify the impact of changes;
- b) *Allowing to watch modifications in files* that contain requirements specifications with the purpose of generating test scripts on demand, automatically. Some testing frameworks (*e.g.*, Mocha, Jest, AVA) provide tools with a similar capability, *i.e.*, they monitor file changes and when a test script file is changed or a file imported by a test script file is changed, they trigger the respective test scripts automatically. This feature can reduce the manual labor to execute tests when specification files change;
- c) *Implementing plug-ins for more testing frameworks*, aiming at fostering the tool's adoption by companies that already use or plan to use them;
- d) *Providing integration with more test reporters*, to help testers to monitor results over time;

Bibliography

- ADWAIT RATNAPARKHI. **A maximum entropy model for part-of-speech tagging**. In Proceedings of the Empirical Methods in Natural Language Processing Conference, 1996.
- ADZIC, Gojko. **Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing**. [s.l.] Neuri Limited, 2009.
- ADZIC, Gojko. **Specification by Example: How Successful Teams Deliver the Right Software**. [s.l.] Manning, 2011.
- AHMED, Bestoun S.; ZAMLI, Kamal Z. **A variable strength interaction test suites generation strategy using Particle Swarm Optimization**. Journal of Systems and Software, v. 84, n. 12, p. 2171–2185, 2011.
- AMBLER, Scott W. **Beyond Functional Requirements On Agile Projects**. Disponível em: <<http://www.drdoobs.com/architecture-and-design/beyond-functional-requirements-on-agile/210601918#>>.
- AMEY, Peter. **Correct by Construction: Better Can Also Be Cheaper**. The Journal of Defense Software Engineering, n. March, 2002.
- ANAND, Saswat; BURKE, Edmund K.; CHEN, Tsong Yueh; CLARK, John; COHEN, Myra B.; GRIESKAMP, Wolfgang; HARMAN, Mark; HARROLD, Mary Jean; MCMINN, Phil. **An orchestrated survey of methodologies for automated software test case generation**. Journal of Systems and Software, v. 86, n. 8, p. 1978–2001, 2013.
- BARR, Earl T.; HARMAN, Mark; MCMINN, Phil; SHAHBAZ, Muzammil; YOO, Shin. **The oracle problem in software testing: A survey**. IEEE Transactions on Software Engineering, v. 41, n. 5, p. 507–525, 2015.
- BASILI, Victor R. **Software modeling and measurement: the Goal/Question/Metric paradigm** Quality, 1992.
- BASRI, S.; O’CONNOR, R. V. **Evaluation on Knowledge Management Process in Very Small Software Companies : A Survey**. Proceedings of Knowledge Management 5th International Conference 2010, 2010.
- BECK, Kent. **Test-Driven Development By Example**. Rivers, v. 2, n. c, p. 176, 2003.
- BEIZER, Boris. **Software Testing Techniques**. [s.l.: s.n.].
- BELL, K. Z. **Optimizing Effectiveness and Efficiency of Software Testing: A Hybrid Approach**. [s.l.] North Carolina State University, 2006.
- BERGIN, Thomas J.; GIBSON, Richard G.; PRESS, A. C. M. **History of Programming Languages II**. [s.l.: s.n.].

- BLASCHEK, G. **Static program analysis**. Elektronische Rechenanlagen, v. 27, n. 2, p. 89–95, 1985.
- BOEHM, B. W. **Verifying and Validating Software Requirements and Design Specifications**. IEEE Software, v. 1, n. 1, p. 75–88, 1984.
- BOEHM, Barry; BASILI, Victor R. **Software Defect Reduction Top 10 List**. Computer, v. 34, p. 135–137, 2001.
- BOEHM, Barry; TURNER, Richard. **Balancing Agility and Discipline: A Guide for the Perplexed**. [s.l.: s.n.]. v. 22
- BOEHM, Barry; TURNER, Richard. **Balancing Agility and Discipline: a guide for the perplexed**. [s.l.] Addison-Wesley, 2003b.
- BOOCH, Robert Grady. **An Economic Release Decision Model: Insights into Software Project Management**. Applications of Software Measurement. Anais...Institute for Software Quality, 1999
- BRIAND, L.; EL EMAM, K.; LAITENBERGER, O.; FUSSBROICH, T. **Using simulation to build inspection efficiency benchmarks for development projects**. Proceedings of the 20th International Conference on Software Engineering. Anais...1998Disponível em: <<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=671387>>
- CALVAGNA, Andrea; GARGANTINI, Angelo. **A logic-based approach to combinatorial testing with constraints**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Anais...2008
- CATAL, Cagatay; MISHRA, Deepti. **Test case prioritization: A systematic mapping study**. Software Quality Journal, v. 21, n. 3, p. 445–478, 2013.
- CHEN, Jingnian; HUANG, Houkuan; TIAN, Shengfeng; QU, Youli. **Feature selection for text classification with Naïve Bayes**. Expert Systems with Applications, v. 36, n. 3 PART 1, p. 5432–5435, 2009a.
- CHEN, Xiang; GU, Qing; LI, Ang; CHEN, Daoxu. **Variable strength interaction testing with an ant colony system approach**. Proceedings - Asia-Pacific Software Engineering Conference, APSEC. Anais...2009b
- CHEN, Xiang; GU, Qing; QI, Jingxian; CHEN, Daoxu. **Applying particle swarm optimization to pairwise testing**. Proceedings - International Computer Software and Applications Conference. Anais...2010
- CHRISTODOULOPOULOS, Christos; STEEDMAN, Mark. **Two Decades of Unsupervised POS induction: How far have we come?** 2010 Conference on Empirical Methods in Natural Language Processing, 2010.
- CIOLKOWSKI, Marcus; LAITENBERGER, Oliver; ROMBACH, Dieter H.; SHULL, Forrest; PERRY, Dewayne. **Software inspections, reviews and walkthroughs**. Proceedings of the 24th international conference on Software engineering - ICSE '02. Anais...2002Disponível em: <<http://dl.acm.org/citation.cfm?id=581339.581422>>
- CIOLKOWSKI, Marcus; LAITENBERGER, Oliver; BIFFL, Stefan. **Software Reviews: The State of the Practice** IEEE Software, 2003.
- COHEN, D. M.; DALAL, S. R.; FREDMAN, M. L.; PATTON, G. C. **The AETG**

- system: an approach to testing based on combinatorial design.** IEEE Transactions on Software Engineering, v. 23, n. 7, p. 437–444, jul. 1997.
- COHEN, Myra B.; DWYER, Matthew B.; SHI, Jiangfan. **Interaction testing of highly-configurable systems in the presence of constraints.** Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07, p. 129, 2007.
- COHN, Mike. **User stories applied: For agile software development.** [s.l: s.n.].
- COHN, Mike. **Non-functional Requirements as User Stories.** Disponível em: <<https://www.mountaingoatsoftware.com/blog/non-functional-requirements-as-user-stories>>.
- COLBOURN, C. J.; KÉRI, G.; RIVAS SORIANO, P. P.; SCHLAGE-PUCHTA, J. C. **Covering and radius-covering arrays: Constructions and classification.** Discrete Applied Mathematics, v. 158, n. 11, p. 1158–1180, 2010.
- COLLINS, Michael. **Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms.** Proceedings of the Conference on Empirical Methods in NLP (EMNLP 2002). Anais...2002
- CURCIO, Karina; NAVARRO, Tiago; MALUCELLI, Andreia; REINEHR, Sheila. **Requirements engineering: A systematic mapping study in agile software development.** Journal of Systems and Software, 2018.
- CUTTING, Doug; KUPIEC, Julian; PEDERSEN, Jan; SIBUN, Penelope. **A practical part-of-speech tagger.** Proceedings of the third conference on Applied natural language processing -. Anais...1992
- CZERWONKA, Jacek. **Pairwise testing in the real world: Practical extensions to test-case scenarios.** Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer. Anais...2006
- DAVIES, Rachel; SEDLEY, Liz. **Agile Coaching.** 1. ed. [s.l.] Pragmatic Bookshelf, 2009.
- DE ALMEIDA FERREIRA, David; DA SILVA, Alberto Rodrigues. **RSLingo: An information extraction approach toward formal requirements specifications.** 2012 2nd IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012 - Proceedings. Anais...2012
- DE MARNEFFE, Marie-Catherine; DOZAT, Timothy; SILVEIRA, Natalia; HAVERINEN, Katri; GINTER, Filip; NIVRE, Joakim; MANNING, Christopher D. **Universal Stanford Dependencies: A cross-linguistic typology.** In Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14), 2014.
- DE MARNEFFE, Marie-Catherine; MANNING, Christopher D. **The Stanford typed dependencies representation.** Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation, 2008.
- DINGSØYR, Torgeir; NERUR, Sridhar; BALIJEPALLY, Venugopal; MOE, Nils Brede. **A decade of agile methodologies: Towards explaining agile software development.** Journal of Systems and Software, v. 85, n. 6, p. 1213–1221, 2012.

- DO, Hyunsook; MIRARAB, Siavash; TAHVILDARI, Ladan; ROTHERMEL, Gregg. **The effects of time constraints on test case prioritization: A series of controlled experiments**. IEEE Transactions on Software Engineering, v. 36, n. 5, p. 593–617, 2010.
- DUBOIS, Catherine; FAMELIS, Michalis; GOGOLLA, Martin; NOBREGA, Leonel; OBER, Ileana; SEIDL, Martina; VÖLTER, Markus. **Research questions for validation and verification in the context of model-based engineering**. CEUR Workshop Proceedings. Anais...2013
- ELBAUM, S.; MALISHEVSKY, A. G.; ROTHERMEL, G. **Test case prioritization: a family of empirical studies**. IEEE Transactions on Software Engineering, v. 28, n. 2, p. 159–182, 2002.
- ELGHONDAKLY, Roaa; MOUSSA, Sherin; BADR, Nagwa. **Waterfall and agile requirements-based model for automated test cases generation**. 2015 IEEE 7th International Conference on Intelligent Computing and Information Systems, ICICIS 2015. Anais...2015
- ENCYCLOPEDIA BRITANNICA. **Concordia**, 2017. (Nota técnica).
- ENDRES, A.; ROMBACH, D. **A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories**. Fraunhofer ed. [s.l.] Pearson/Addison Wesley, 2003.
- ERNST, Neil A.; BORGIDA, Alexander; JURETA, Ivan J.; MYLOPOULOS, John. **Agile requirements engineering via paraconsistent reasoning**. Information Systems, 2014.
- EVANS, Eric. **Domain-Driven Design: Tackling Complexity in the Heart of Software**. [s.l.] Addison-Wesley Professional, 2003.
- FAGAN, M. E. **Design and code inspections to reduce errors in program development**. IBM Systems Journal, v. 15, n. 3, p. 182–211, 1976.
- FINOT, Olivier; MOTTU, Jean Marie; SUNYÉ, Gerson; ATTIOGBÉ, Christian. **Partial Test Oracle in Model Transformation Testing**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Anais...2013
- FOWLER, Martin. **Business-Readable DSL**. Disponível em: <<https://martinfowler.com/bliki/BusinessReadableDSL.html>>.
- FOWLER, Martin. **Domain-Specific Languages**. [s.l.] Addison-Wesley Professional, 2009. v. 5658
- GAIKWAD, Vandana; JOEG, Prasanna. **An Empirical Study of Writing Effective User Stories**. International Journal of Software Engineering and Its Applications, v. 10, n. 11, p. 387–404, 2016.
- GÄRTNER, Markus. **ATDD by Example: a practical guide to Acceptance Test-Driven Development**. [s.l.] Addison-Wesley, 2012.
- GOODGER, David. **reStructuredText**. Disponível em: <<http://docutils.sourceforge.net/rst.html>>.
- GOPNIK, Alison; SLAUGHTER, Virginia; MELTZOFF, Andrew N. **Changing your views: how understanding visual perception can lead to a new theory of the mind**. In: Children's Early Understanding of Mind: Origins and

Development. [s.l: s.n.].

GREGORY, Janet; CRISPIN, Lisa. **ATDD vs. BDD vs. Specification by Example vs** Disponível em: <<http://janetgregory.ca/atdd-vs-bdd-vs-specification-by-example-vs/>>. Acesso em: 1 ago. 2017.

GRINDAL, Mats; OFFUTT, Jeff; ANDLER, Sten F. **Combination testing strategies: A survey**. Software Testing Verification and Reliability, 2005.

GRUBER, John. **Markdown**. Disponível em: <<https://daringfireball.net/projects/markdown/>>.

HALLE, Barbara Von. **Business Rules Applied — Business Better Systems Using the Business Rules Approach**. [s.l: s.n.].

HARTMAN, Alan. **Software and Hardware Testing Using Combinatorial Covering Suites**. In: Graph Theory Combinatorics and Algorithms. [s.l: s.n.]. v. 34p. 237–266.

HELLESØY, Aslak. **Cucumber**. Disponível em: <<https://cucumber.io/>>.

HENDRICKSON, Elisabeth. **Acceptance Test Driven Development (ATDD): an Overview**. Disponível em: <<http://testobsessed.com/2008/12/acceptance-test-driven-development-atdd-an-overview/>>.

HENDRICKSON, Elisabeth. **Driving development with tests: ATDD and TDD**. Quality Tree Software, Inc. <http://www.qualitytree.com>, p. 1–9, 2008b.

HOLTMANN, Jörg; MEYER, Jan; VON DETTEN, Markus. **Automatic validation and correction of formalized, textual requirements**. Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011. Anais...2011

INAYAT, Irum; SALIM, Siti Salwah; MARCZAK, Sabrina; DANEVA, Maya; SHAMSHIRBAND, Shahaboddin. **A systematic literature review on agile requirements engineering practices and challenges** Computers in Human Behavior, 2015.

ISO; IEC; IEEE. **Systems and software engineering -- Developing user documentation in an agile environment** ISO/IEC/IEEE 26515 First edition 2011-12-01; Corrected version 2012-03-15, 2012.

ISO; IEC; IEEE. **ISO/IEC 26550:2015 Software and systems engineering — Reference model for product line engineering and managemen**. [s.l: s.n.].

ISO; IEC; IEEE. **ISO/IEC/IEEE 29119-5:2016 - Software and systems engineering -- Software testing -- Part 5: Keyword-Driven Testing**.

ISO; IEC; IEEE. **ISO/IEC/IEEE 24765:2017 - Systems and software engineering - Vocabulary**. [s.l: s.n.].

JÉZÉQUEL, Jean Marc; MÉNDEZ-ACUÑA, David; DEGUEULE, Thomas; COMBEMALE, Benoît; BARAIS, Olivier. **When systems engineering meets software language engineering**. Complex Systems Design and Management - Proceedings of the 5th International Conference on Complex Systems Design and Management, CSD and M 2014. Anais...2015

JOACHIMS, Thorsten. **Text Categorization with Suport Vector Machines: Learning with Many Relevant Features**. Proceedings of the 10th European

- Conference on Machine Learning ECML '98, p. 137–142, 1998.
- JONES, Capers. **Software defect-removal efficiency**. Computer, v. 29, n. 4, p. 94–95, 1996.
- JONES, Capers. **Estimating Software Costs**. New York, NY, USA: McGraw-Hill, 1998.
- JONES, Capers. **Evaluating Software Metrics and Software Measurement Practices**. Namcook Analytics, 2014.
- JONES, Capers; BONSIGNOUR, Olivier. **The Economics of Software Quality**. 1st. ed. [s.l.] Addison-Wesley, 2012.
- JURAFSKY, Daniel; MARTIN, James H. **Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition**. Speech and Language Processing An Introduction to Natural Language Processing Computational Linguistics and Speech Recognition, 2009.
- KAMALAKAR, Sunil; EDWARDS, Stephen H.; DAO, Tung M. **Automatically generating tests from natural language descriptions of software behavior**. ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering. Anais...2013Disponível em: <<http://www.scopus.com/inward/record.url?eid=2-s2.0-84887061776&partnerID=tZOtx3y1>>
- KELLEY, Richard; TAVAKKOLI, Alireza; KING, Christopher; AMBARDEKAR, Amol; NICOLESCU, Monica; NICOLESCU, Mircea. **Context-based Bayesian intent recognition**. IEEE Transactions on Autonomous Mental Development, 2012.
- KOSAR, Tomaž; BOHRA, Sudev; MERNIK, Marjan. **Domain-Specific Languages: A Systematic Mapping Study**. Information and Software Technology, v. 71, p. 77–91, 2016.
- KUHN, D. R.; REILLY, M. J. **An investigation of the applicability of design of experiments to software testing**. Proceedings - 27th Annual NASA Goddard / IEEE Software Engineering Workshop, SEW 2002. Anais...2003
- KUHN, D. R.; WALLACE, D. R.; GALLO, A. M. **Software fault interactions and implications for software testing**. IEEE Transactions on Software Engineering, v. 30, n. 6, p. 418–421, 2004.
- LAFFERTY, John; MCCALLUM, Andrew; PEREIRA, Fernando C. N. **Conditional random fields: Probabilistic models for segmenting and labeling sequence data**. ICML '01 Proceedings of the Eighteenth International Conference on Machine Learning, 2001.
- LAITENBERGER, Oliver. **A survey of software inspection technologies**. In: Handbook on Software Engineering and Knowledge Engineering. 2. ed. [s.l.: s.n.], p. 517–555.
- LANDI, William. **Undecidability of static analysis**. ACM Letters on Programming Languages and Systems, v. 1, n. 4, p. 323–337, 1992.
- LAUKKANEN, Pekka. **Data-Driven and Keyword-Driven Test Automation Frameworks**. [s.l.: s.n.].

- LEE, Allen S.; BASKERVILLE, Richard L. **Generalizing Generalizability in Information Systems Research**. Information Systems Research, 2003.
- LEFFINGWELL, Dean. **Calculating the return on investment from more effective requirements management**. Cutter IT Journal, v. 10, n. 4, p. 13–16, 1997.
- LEFFINGWELL, Dean. **Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise** Pearson Education Inc, 2011.
- LEI, Y.; TAI, K. C. **In-parameter-order: a test generation strategy for pairwisetesting**. IEEE International High-Assurance Systems Engineering Symposium. Anais...1998
- LEI, Yu; KACKER, Raghu; KUHN, D. Richard; OKUN, Vadim; LAWRENCE, James. **IPOG-IPOG-D: Efficient test generation for multi-way combinatorial testing**. Software Testing Verification and Reliability, v. 18, n. 3, p. 125–148, 2008.
- LIU, Huai; KUO, Fei-Ching; TOWEY, D.; CHEN, Tsong Yueh. **How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?** IEEE Transactions on Software Engineering, v. 40, n. 1, p. 4–22, 2014.
- MAALEM, Sourour; ZAROOUR, Nacereddine. **Challenge of validation in requirements engineering**. Journal of Innovation in Digital Ecosystems, 2016.
- MANNING, Christopher D.; RAGHAVAN, Prabhakar; SCHÜTZE, Hinrich. **Introduction to Information Retrieval**. 1st. ed. [s.l.] Cambridge University Press, 2008.
- MARCUS, Mitchell P.; SANTORINI, Beatrice; MARCINKIEWICZ, Mary Ann. **Building a large annotated corpus of English: The Penn Treebank**. Computational Linguistics, 1993.
- MARTIN, J. **Fourth-Generation Languages**. Vol. I: Pr ed. [s.l.] Prentice-Hall, 1985.
- MCCALLUM, Andres; NIGAM, Kamal. **A Comparison of Event Models for Naive Bayes Text Classification**. AAAI/ICML-98 Workshop on Learning for Text Categorization, p. 41–48, 1998.
- MCCALLUM, Andrew; FREITAG, Dayne; PEREIRA, Fernando. **Maximum Entropy Markov Models for Information Extraction and Segmentation**. Proceedings of the Seventeenth International Conference on Machine Learning. Anais...2000
- MCCONNELL, Steve. **Code Complete: A Practical Handbook of Software Construction**. Design, p. 919, 2004.
- MERNIK, Marjan; HEERING, Jan; SLOANE, Anthony M. **When and how to develop domain-specific languages**. ACM Computing Surveys, v. 37, n. 4, p. 316–344, 2005.
- MEYER, David. **Support Vector Machines**. R News, 2001.
- MILL, Harlan D.; WEINBERG, Gerald M. **Software Productivity**. [s.l.] Dorset House Publishing Company, 1988.

- MILLER, George A. **WordNet: a lexical database for English**. Communications of the ACM, 1995.
- MOHANTY, Sanjukta; ACHARYA, Arup Abhinna; MOHAPATRA, Durga Prasad. **A Survey on Model-Based Test Case Prioritization**. International Journal of Computer Science and Information Technologies. Anais...2011
- MORALES-TRUJILLO, Miguel; OKTABA, Hanna; PIATTINI, Mario. **Using Technical-Action-Research to Validate a Framework for Authoring Software Engineering Methods**. Proceedings of the 17th International Conference on Enterprise Information Systems, 2015.
- MORALI, Ayşe; WIERINGA, Roel. **Risk-based confidentiality requirements specification for outsourced IT systems**. Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, RE2010. Anais...2010
- MYAENG, Sung Hyon; HAN, Kyoung Soo; RIM, Hae Chang. **Some effective techniques for naive bayes text classification**. IEEE Transactions on Knowledge and Data Engineering, v. 18, n. 11, p. 1457–1466, 2006.
- MYERS, Glenford J.; THOMAS, Todd M.; SANDLER, Corey. **The Art of Software Testing 3rd Edition**. [s.l: s.n.]. v. 1
- NARDI, Bonnie A. **A Small Matter of Programming: Perspectives on End User Computing**. [s.l: s.n.]. v. 26
- NIE, Changhai; LEUNG, Hareton. **A survey of combinatorial testing**. ACM Computing Surveys, v. 43, n. 2, p. 1–29, 2011.
- NOKIA CORPORATION. **Robot Framewok**. Disponível em: <<http://robotframework.org/>>.
- NORTH, Dan. **JBehave**. Disponível em: <<http://jbehave.org/>>.
- NORTH, Dan. **Introducing BDD** Better Software Magazine, 2006. Disponível em: <<http://dannorth.net/introducing-bdd/>>
- NURMELA, Kari J. **Upper bounds for covering arrays by tabu search**. Discrete Applied Mathematics. Anais...2004
- PARRA, Otto; ESPANA, Sergio; PANACH, Jose Ignacio. **Extending and validating gestUI using technical action research**. Proceedings - International Conference on Research Challenges in Information Science. Anais...2017
- PINTO, Thiago Delgado; STAA, Arndt Von. **Uma Ferramenta para Geração e Execução Automática de Testes Funcionais Baseados na Descrição Textual de Casos de Uso**. [s.l.] Pontifical Catholic University of Rio de Janeiro (PUC-Rio), 2013.
- POPPENDIECK, Mary. **Lean software development**. 29th International Conference on Software Engineering (ICSE). Anais...IEEE, 2007Disponível em: <<https://www.computer.org/csdl/proceedings/icsecompanion/2007/2892/00/28920165.pdf>>
- PORTER, Martin F. **An algorithm for suffix stripping**. Program, v. 14, n. 3, p. 130–137, 1980.

- PREMACK, David; WOODRUFF, Guy. **Does the chimpanzee have a theory of mind?** Behavioral and Brain Sciences, 1978.
- PUGH, Ken. **Lean-Agile Acceptance Test-Driven Development: better software through collaboration.** [s.l.] Addison-Wesley, 2011.
- R. GLASS; VESSEY, I.; RAMESH, V. **Research in software engineering: an empirical study.** [s.l: s.n.].
- RANE, Prerana Pradeepkumar. **Automatic Generation of Test Cases for Agile using Natural Language Processing.** [s.l.] Faculty of the Virginia Polytechnic Institute and State University, 2017.
- RASJID, Zulfany Erlisa; SETIAWAN, Reina. **Performance Comparison and Optimization of Text Document Classification using k-NN and Naïve Bayes Classification Techniques.** Procedia Computer Science, v. 116, p. 107–112, 2017.
- ROBEER, Marcel; LUCASSEN, Garm; VAN DER WERF, Jan Martijn E. M.; DALPIAZ, Fabiano; BRINKKEMPER, Sjaak. **Automated Extraction of Conceptual Models from User Stories via NLP.** Proceedings - 2016 IEEE 24th International Requirements Engineering Conference, RE 2016. Anais...2016
- ROBINSON, W. S. **The logical structure of analytic induction.** American Sociological Review, 1951.
- ROSA, Otávio Araújo Leitão. **Test-Driven Maintenance: uma abordagem para manutenção dos sistemas legados.** [s.l.] Pontifical Catholic University of Rio de Janeiro, 2011.
- ROTHERMEL, Gregg; UNTCH, Roland H.; CHU, Chengyun; HARROLD, Mary Jean. **Test Case Prioritization: an Empirical Study.** Proceedings of the IEEE International Conference on Software Maintenance, p. 179, 1999.
- RUNESON, Per; HOST, Martin; RAINER, Austen; REGNELL, Björn. **Case Study Research in Software Engineering.** [s.l: s.n.].
- SAMMET, Jean E. **Programming languages: history and fundamentals.** [s.l.] Prentice-Hall, 1969.
- SANG, Erik F. Tjong Kim; DE MEULDER, Fien. **Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition.** CONLL '03 Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003, 2003.
- SANTORINI, Beatrice. **Part-of-Speech Tagging Guidelines for the Penn Treebank Project (3rd Revision).** University of Pennsylvania 3rd Revision 2nd Printing, 1990.
- SCHAPIRE, Robert E.; SINGER, Yoram. **BoosTexter: A Boosting-based System for Text Categorization.** Machine Learning, v. 39, p. 135–168, 2000.
- SCHÖN, Eva-Maria; THOMASCHEWSKI, Jörg; ESCALONA, María José. **Agile Requirements Engineering: A systematic literature review.** Computer Standards & Interfaces, 2017.
- SCHWABER, K. **Agile Project Management with Scrum.** [s.l: s.n.]. v. 7

- SEDDON, Peter B.; SCHEEPERS, Rens. **Towards the improved treatment of generalization of knowledge claims in IS research: Drawing general conclusions from samples**. European Journal of Information Systems, 2012.
- SEDDON, Peter; SCHEEPERS, Rens. **Other-Settings Generalizability in IS Research**. ICIS. Anais...2006
- SHERWOOD, George. **Effective testing of factor combinations**. Proc. Third International Conference on Software Testing, Analysis and Review (STAR'94). Anais...1994
- SHIBA, T.; TSUCHIYA, T.; KIKUNO, T. **Using artificial life techniques to generate test cases for combinatorial testing**. Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004., p. 1–14, 2004.
- SHULL, F.; BASILI, V.; BOEHM, B.; BROWN, A. W.; COSTA, P.; LINDVALL, M.; PORT, D.; RUS, I.; TESORIERO, R.; ZELKOWITZ, M.; ALLEN, Ed; ANGER, Frank; CHULANI, Sunita; DAVIS, Noopur; DYER, Michael; EBERT, Christof; ELLIOTT, Bill; FAGAN, Eileen; FEATHER, Martin; GREEN, Liz; FORMAN, Ira; HENNINGER, Scott; JOHNSON, Philip; LAITENBERGER, Oliver; MADACHY, Ray; MATSUMOTO, Yoshihiro; MCGIBBON, Tom; MILLER, James; MOORE, James; O'NEILL, Don; RIFKIN, Stan; ROMBACH, Dieter; ROY, Dan; SAIEDIAN, Hossein; SUCCI, Giancarlo; THOMAS, Gary; VINTER, Otto. **What we have learned about fighting defects**. Proceedings - International Software Metrics Symposium. Anais...2002
- SMART, John Ferguson. **BDD in action**. [s.l.] Manning Publications, 2014.
- SOEKEN, Mathias; WILLE, Robert; DRECHSLER, Rolf. **Assisted Behaviour Driven Development Using Natural Language Processing**. 50th International Conference on Objects, Models, Components, Patterns (TOOLS), 2012.
- SOMMERVILLE, Ian. **Software Engineering**. 9th. ed. Boston, MA, USA: Pearson, 2011.
- SRIKANTH, Hema; HETTIARACHCHI, Charitha; DO, Hyunsook. **Requirements based test prioritization using risk factors: An industrial study**. Information and Software Technology, v. 69, p. 71–83, 2016.
- SRIKANTH, Hema; WILLIAMS, Laune; OSBORNE, Jason. **System test case prioritization of new and regression test cases**. 2005 International Symposium on Empirical Software Engineering, ISESE 2005. Anais...2005
- STAA, Arndt Von. **Lecture Notes on Software Testing (INF1413) - Software Quality - Concepts (in Portuguese)**Rio de JaneiroPontifical Catholic University of Rio de Janeiro, , 2017. Disponível em: <www.inf.puc-rio.br/~inf1413/>
- STAVRU, Stavros. **A critical examination of recent industrial surveys on agile method usage**. Journal of Systems and Software, v. 94, p. 87–97, 2014.
- TACQ, Jacques. **Znaniecki's analytic induction as a method of sociological research**. Polish Sociological Review. Anais...2007
- TAYLOR, J.; MAZLACK, L. **Toward computational recognition of humorous**

intent. Proceedings of the Annual Conference of the Cognitive Science Society, 2005.

TELEFÓNICA. **Tartare.** Disponível em:
<<https://github.com/telefonicaid/tartare>>.

THE CUCUMBER TEAM. **Gherkin.** Disponível em:
<<https://github.com/cucumber/cucumber/wiki/Gherkin>>.

THOUGHTWORKS. **Gauge.** 2014.

TUNG, Yu-Wen; ALDIWAN, W. S. **Automating test case generation for the new generation mission software system.** Proceedings of the IEEE Aerospace Conference, v. 1, p. 431–437, 2000.

VERMA, Ravi Prakash; BEG, Md. Rizwan. **Generation of Test Cases from Software Requirements Using Natural Language Processing.** 2013 6th International Conference on Emerging Trends in Engineering and Technology. Anais...IEEE, dez. 2013Disponível em:
<<http://ieeexplore.ieee.org/document/6754807/>>. Acesso em: 2 fev. 2017

VIDIEMME CONSULTING. **Bravey.**

VIDYA SAGAR, Vidhu Bhala R.; ABIRAMI, S. **Conceptual modeling of natural language functional requirements.** Journal of Systems and Software, v. 88, n. 1, p. 25–41, 2014.

WALLACE D R, Kuhn D. R. **Failure modes in medical device software: An analysis of 15 years of recall data.** International Journal of Reliability Quality and Safety Engineering, 2001.

WAUTELET, Yves; HENG, Samedi; KOLP, Manuel; MIRBEL, Isabelle. **Unifying and extending user story models.** Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Anais...2014

WEXELBLAT, Richard L. **History of Programming Languages (ACM SIGPLAN).** ACM SIGPLAN History of Programming Languages. Anais...1981Disponível em:
<<http://dl.acm.org.proxy.library.cornell.edu/citation.cfm?id=1198340&bnc=1>>

WEYUKER, Elaine J. **On testing non-testable programs.** Computer Journal, v. 25, n. 4, p. 465–470, 1982.

WHEELER, David A.; BRYKCZYNSKI, Bill; MEESON JR, Reginald N. **Software Inspection: An Industry Best Practice for Defect Detection and Removal.** [s.l.] IEEE Computer Society Press Los Alamitos, 1996.

WIEGERS, Karl. **Peer Reviews in Software: A practical guide.** Boston, MA, USA: Addison-Wesley, 2002.

WIEGERS, Karl E.; BEATTY, Joy. **Software Requirements.** [s.l.: s.n.].

WIERINGA, Roel. **Design Science Methodology for Information Systems and Software Engineering.** [s.l.] Springer, 2014a.

WIERINGA, Roel. **Empirical research methods for technology validation: Scaling up to practice.** Journal of Systems and Software, 2014b.

- WIERINGA, Roel; DANEVA, Maya. **Six strategies for generalizing software engineering theories**. Science of Computer Programming. Anais...2015
- WIERINGA, Roel; DANEVA, Maya; CONDORI-FERNANDEZ, Nelly. **The Structure of Design Theories, and an Analysis of their Use in Software Engineering Experiments**. 2011 International Symposium on Empirical Software Engineering and Measurement. Anais...2011
- WIERINGA, Roel J. **Technical Action Research**. Design Science Methodology for Information Systems and Software Engineering, 2014c.
- WIKIPEDIA. **Behavior-Driven Development**. Disponível em: <https://en.wikipedia.org/wiki/Behavior-driven_development>. Acesso em: 26 nov. 2017.
- WILLIAMS, Alan Webber. **Software component interaction testing: Coverage measurement and generation of configurations**. [s.l.] University of Ottawa (Canada), 2002.
- WYNNE, Matt; HELLESØY, Aslak. **The Cucumber Book: Behaviour-Driven Development for Testers and Developers**. [s.l.] Pragmatic Bookshelf, 2012.
- YAN, Jun; ZHANG, Jian. **A backtracking search tool for constructing combinatorial test suites**. Journal of Systems and Software, v. 81, n. 10, p. 1681–1693, 2008.
- YANG, Yiming; CHUTE, Christopher G. **An example-based mapping method for text categorization and retrieval**. ACM Transactions on Information Systems, v. 12, n. 3, p. 252–277, 1994.
- YIN, Robert K. **Case Study Research . Design and Methods** SAGE Publications, 2003.
- YOO, S.; HARMAN, M. **Regression testing minimization, selection and prioritization: A survey** Software Testing Verification and Reliability, 2012.
- YOUSUF, Farzana; ZAMAN, Zahid; IKRAM, Naveed. **Requirements validation techniques in GSD: A survey**. IEEE INMIC 2008: 12th IEEE International Multitopic Conference - Conference Proceedings. Anais...2008
- YUE, Tao; BRIAND, Lionel C.; LABICHE, Yvan. **A systematic review of transformation approaches between user requirements and analysis models** Requirements Engineering, 2011.
- ZELKOWITZ, Marvin V.; WALLACE, Dolores. **Experimental validation in software engineering**. Information and Software Technology, 1997.
- ZHENG, J.; WILLIAMS, L.; NAGAPPAN, N.; SNIPES, W.; HUDEPOHL, J. P.; VOUK, M. **a Software Engineering Ieee Transactions On. On the value of static analysis for fault detection in software**. Software Engineering, IEEE Transactions on, v. 32, n. 4, p. 240–253, 2006.
- ZNANIECKI, F. **The Method of Sociology**. 2nd editio ed. [s.l.] Octagon Books, 1968.

Appendix A – Architecture of the Solution

Simple things should be simple; complex things should be possible.

- Alan Kay

This appendix details the architecture of the solution that reifies our approach.

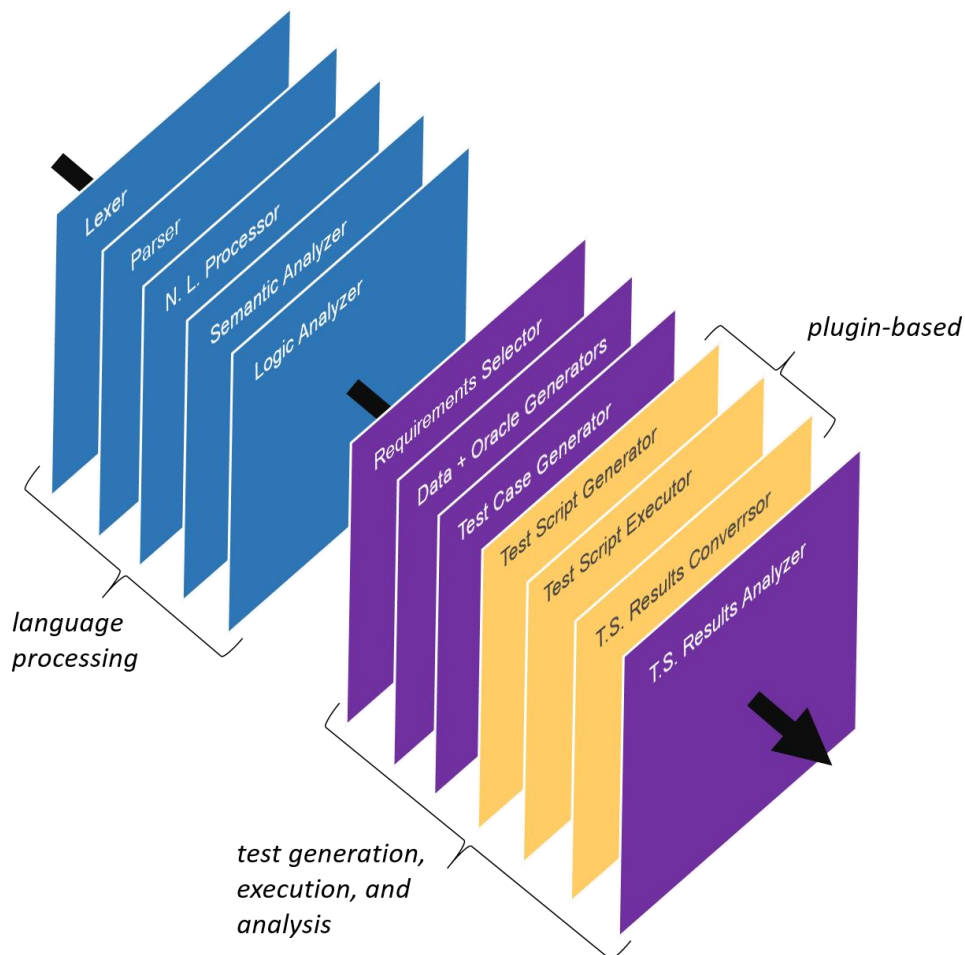


Figure A1 - Processing Stages

Figure A1 shows the stages that can be followed by a tool that implements our approach. *Input* is composed of a set of text documents written in Concordia and a set of execution parameters. *Output* is composed of messages describing the processing results and a set of files that includes test scripts (one test script file per

feature file), test script configuration (a single file), and test script execution results (one file per test execution or a single file).

The first group of stages performs the *language processing*. *Lexer* detects sequences of characters that match a pattern – called *lexemes* – and labels these lexemes, forming *tokens*. *Parser* receives these tokens, checks their syntax, and constructs an *abstract syntax tree* (AST). The *Natural Language Processor* receives the AST and analyzes its syntax deeply aiming to augment it and, thus, to create an *extended abstract syntax tree* (EAST). *Semantic Analyzer* checks properties and relations among tokens in the EAST, which include their scopes, types, parameters, and references. Information about detected properties and relations are attached to the EAST, creating an *annotated abstract syntax tree* (AAST). Finally, the *Logic Analyzer* checks the AAST for logic conflicts in declarations, such as constraints with conflicting value ranges and cyclic references. The AAST is then ready to be processed by the test case generator.

The second group of stages addresses the generation, execution, and analysis of test cases. *Requirements Selector* filters the AAST according to the parameterized selection strategy – we describe the strategies over the section 7.3. Test scenarios are generated from the filtered AAST, along with test data and test oracles, and then transformed into Test Cases. The Test Cases are exported to files with the extension `.testcase`, and transformed into Abstract Test Scripts – *i.e.*, a format simpler to be processed by plugins. The next three steps are executed by a single plug-in: Test Script generator transforms abstract test scripts into test scripts (source code); Test Script Executor executes the test scripts; Test Script Results Converter reads execution results and transforms them into a format that Concordia can understand. Finally, the transformed results are analyzed and reported to the user.

The next sections give more details about the structure, with the help of the Unified Modeling Language (UML).

A1. Lexer

Figure A2 shows the structure of the lexer (class `Lexer`). Since Concordia has line-based declarations, new lines can be recognized from the method `addNodeFromLine`. A node represents a declaration. The lexer uses node lexers (interface `NodeLexer`) for detecting different node types – there are more than 30 of them. These node lexers are instantiated when the lexer is created. Every node lexer has a specific purpose, such as identifying tags (class `TagNodeLexer`), identifying constants (class `ConstantNodeLexer`), etc., and is responsible for suggesting node types that are often declared after it, to speed up the detection.

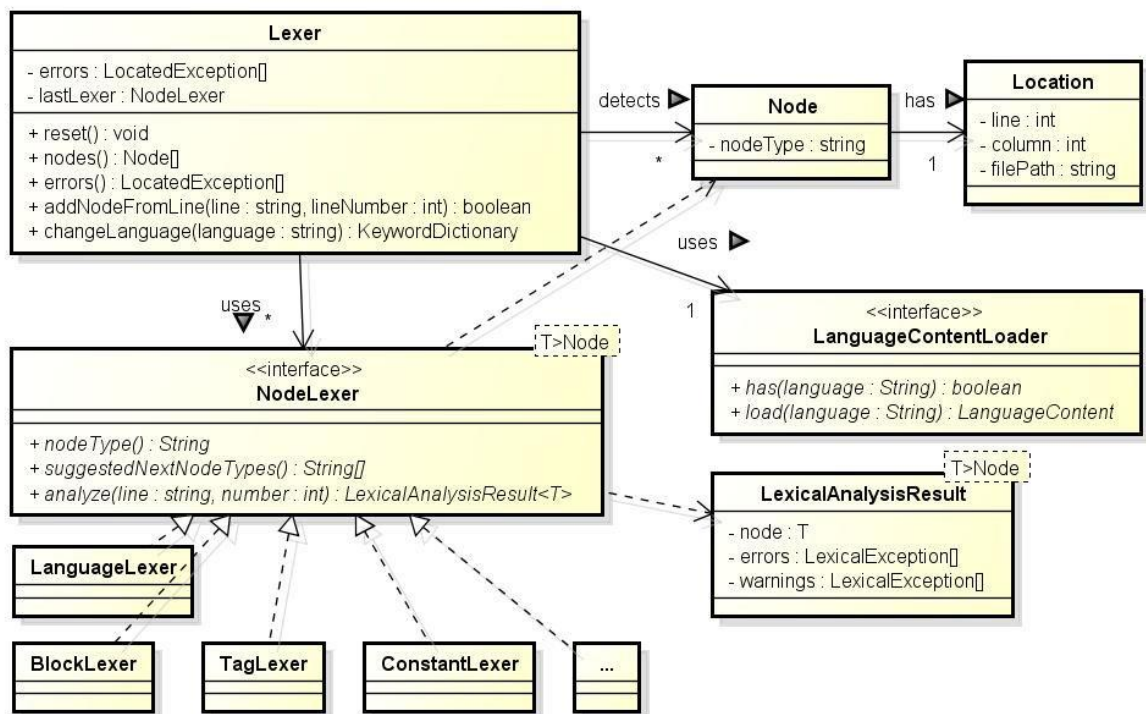


Figure A2 - Lexer

Figure A3 presents the structure of the language loader, used by the lexer. When a document contains a language declaration (section 6.1.2), e.g., “`#language: pt`”, the lexer automatically loads the language content (class `LanguageContent`) using a loader (class `LanguageContentLoader`). That content contains the vocabulary used in DSLs, Intents, Entities, training examples, and data test case names.

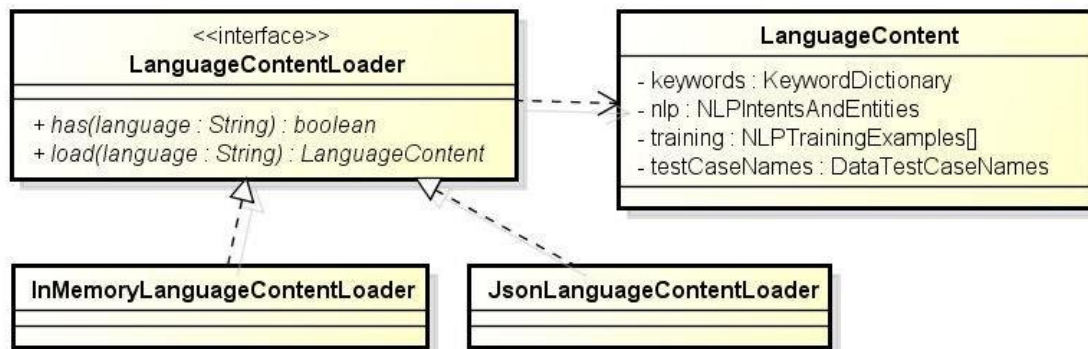


Figure A3 - Language Content Loader

A2. Parser

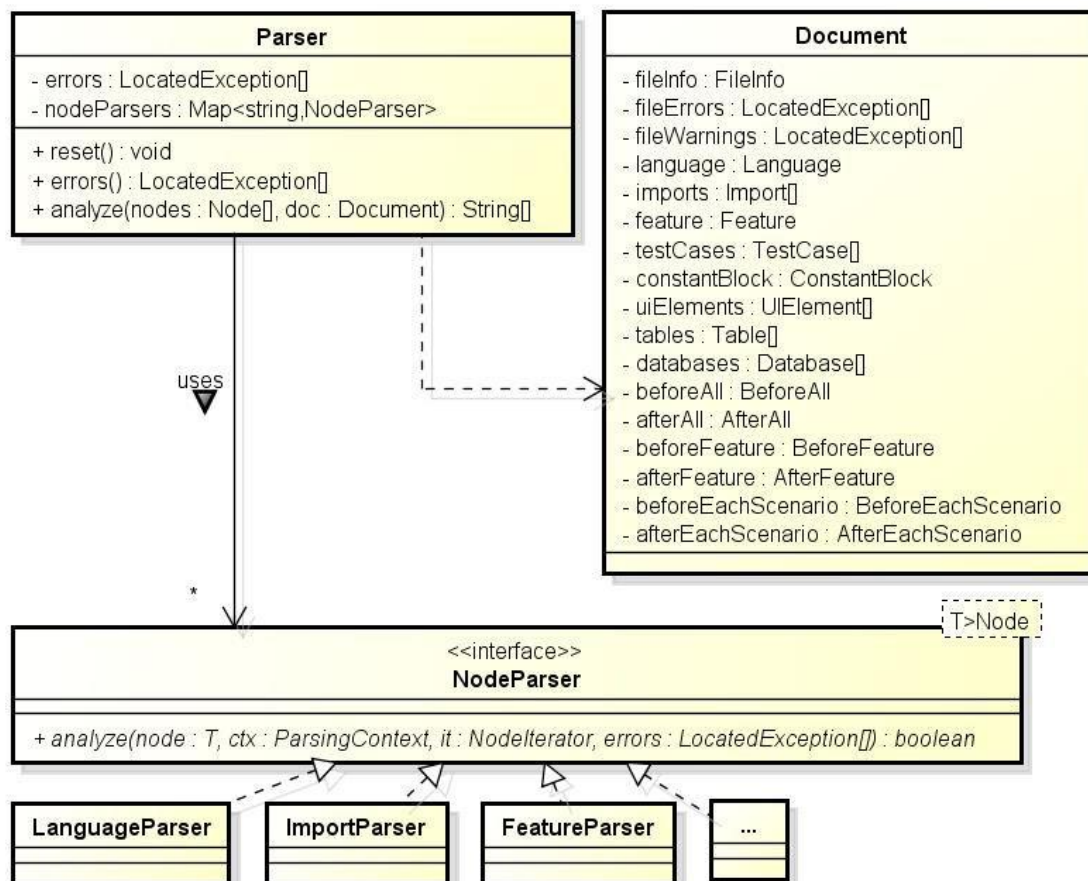


Figure A4 - Parser

Figure A4 shows the structure of the parser (class Parser). It uses specialized node parsers (interface NodeParser) for analyzing the given nodes and putting them into the given document (class Document), which serves as an abstract syntax tree.

Figure A5 represents the specification (class Spec) and its documents (class Document).

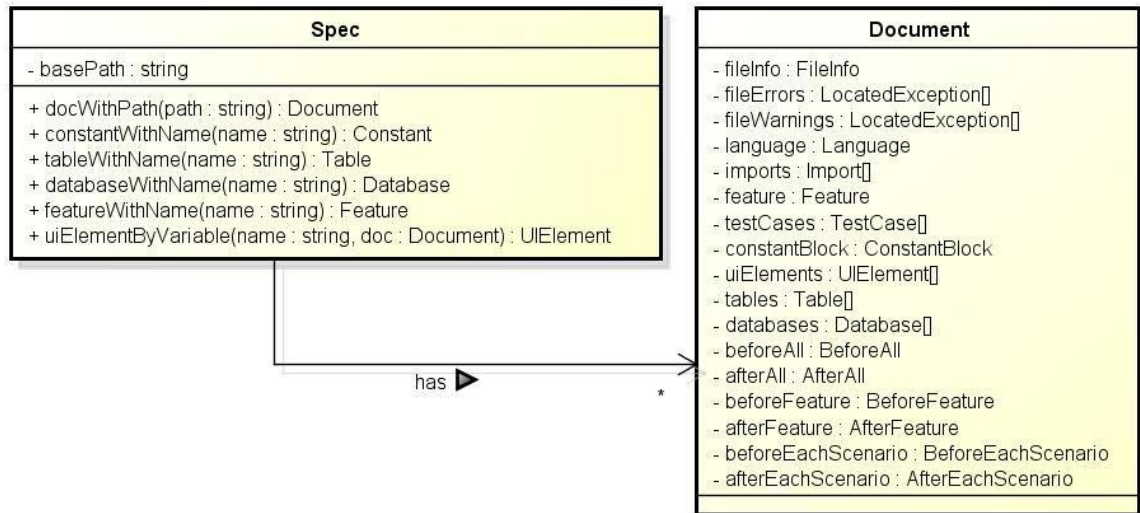


Figure A5 - Specification

A3. Natural language processor

Figure A6 shows the basic structure adopted for using a natural language processor. The interface `NLPStrategy` can have different implementations in order to support alternative techniques – such as those mentioned in section 5.3, *e.g.*, Supporting Vector Machines, Conditional Random Field, Averaged Perceptron, Hidden Markov Model. As mentioned in section 5.4, we chose an implementation based on a Naïve-Bayes Classifier called Bravey (used by the class `BraveyStrategy`). Future research directions may include the comparison of such implementations. We detailed how we used Bravey in section 5.6.

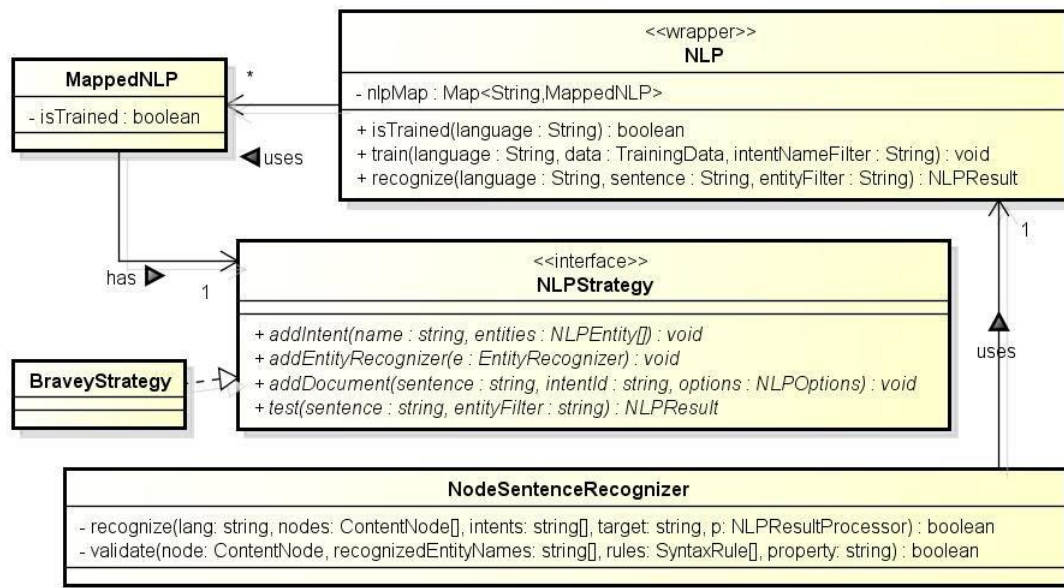


Figure A6 - Structure for NLP

The class NLP provides a high-level interface for training and recognizing sentences for a certain spoken language. Training data is loaded from a dictionary, created for every supported spoken language. The class NodeSentenceRecognizer uses the class NLP for recognizing sentences of parsed nodes (resulting from the parsing process) and can validate sentences using the syntax rules (class SyntaxRule) defined for the expected intents. Warnings and errors detected (we omitted them from the parameters of the method validate) contain their locations in these sentences.

A4. Semantic and logic analyzers

For performance reasons, semantic analysis and logic analysis are executed by the same classes. We differentiated the analysis of a single document from the analysis of the entire specification. Figure A7 shows the structure of a *document* analyzer (interface DocumentAnalyzer), while Figure A8 shows the structure of a *specification* analyzer (interface SpecificationAnalyzer). Both have implementations that vary according to the node types. For example, the class ImportDA (an implementation of DocumentAnalyzer) analyzes duplicated imports, self-references, and the existence of the declared import files. The class ImportSSA (an implementation of SpecificationAnalyzer) checks for cyclic references. We exemplify the analysis performed by our approach in section 8.2. All these analyzers are

executed in a specific order, in batch. Specification analysis occurs only after all the documents have been analyzed individually.

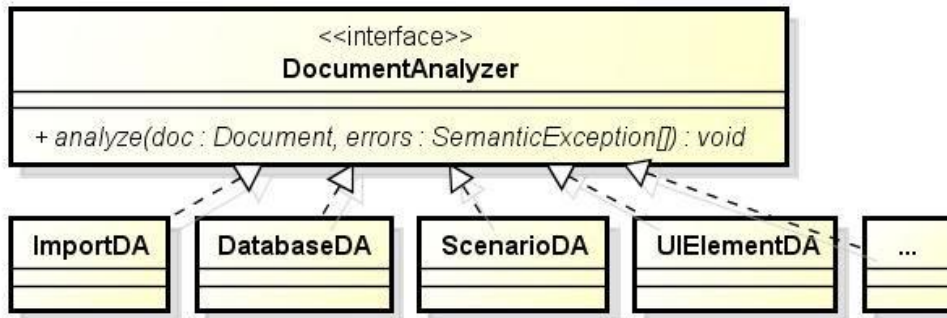


Figure A7 - Document Analyzer

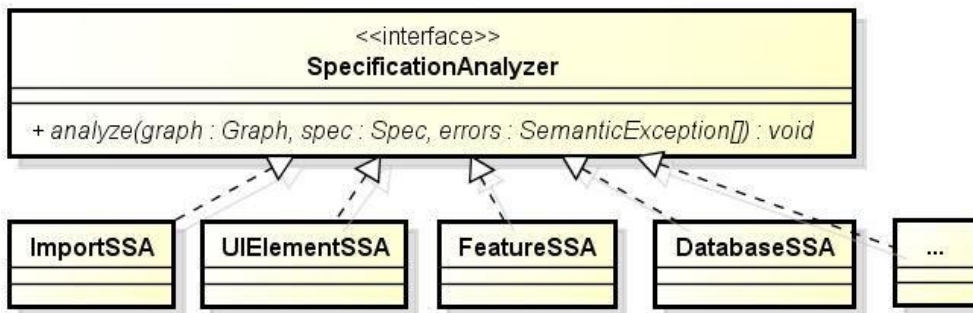


Figure A8 - Specification Analyzer

A5. Test scenario generator

Figure A9 represents the combination strategies explained in section 7.3.3. As mentioned before, a combination strategy (*i.e.*, an implementation of the interface **CombinationStrategy**) can be used in different moments in the test case generation.

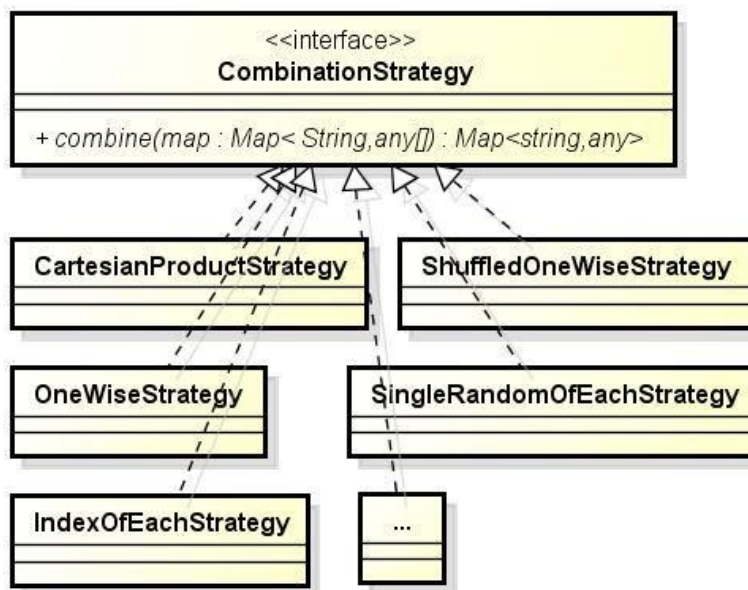


Figure A9 - Combination Strategies

Figure A10 represents the variant selection strategies (interface `VariantSelectionStrategy`) used in the test scenario generation (Figure A11). Section 7.3.4.1 explains these strategies' approaches.

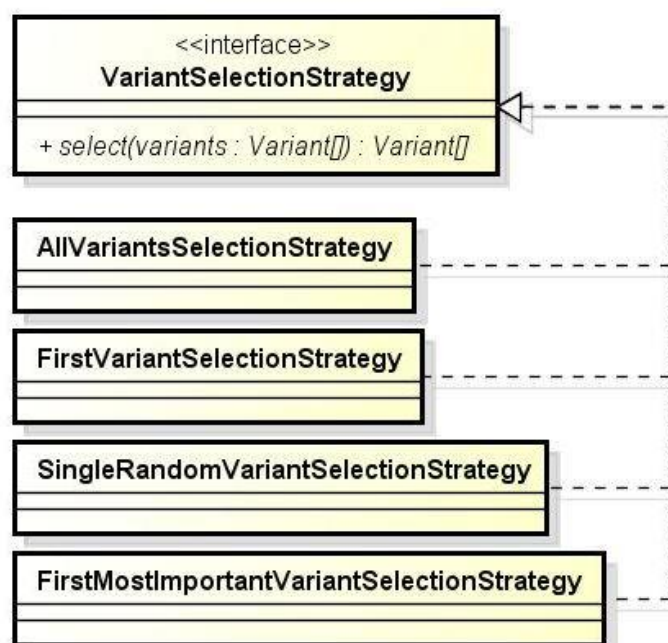


Figure A10 - Variant Selection Strategy

Figure A11 represents the test scenario generator (class `TestScenarioGenerator`). It generates test scenarios (class `TestScenario`) for a certain `Variant` and maps these scenarios (class `TSMaps`) to facilitate their combination with other test scenarios. The generator uses a strategy to select `Variants` (interface `VariantSelectionStrategy`) that produces certain `States`, and use a strategy to combine `Test Scenarios` of these `Variants` (interface `CombinationStrategy`).

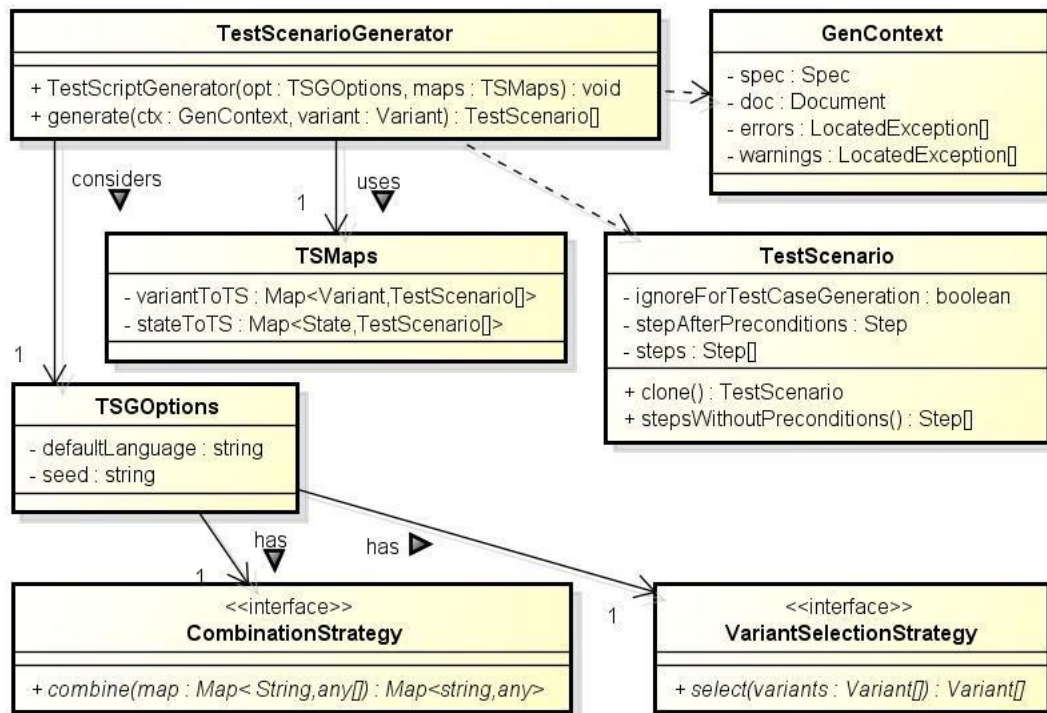


Figure A11 - Test Scenario Generator

A6. Test data and test oracle generators

As we explain in section 7.3.5, the adopted mix of data tests has a direct impact on the number of produced test cases, on their capability to detect defects, on their oracles, and on their behavior. Users may adopt the mix that fits better their systems, time, and rigor for testing. Figure A12 represents an interface to mix data test cases, `DataTestCaseMix`, whose approaches were detailed in Table 17 (section 7.3.5).

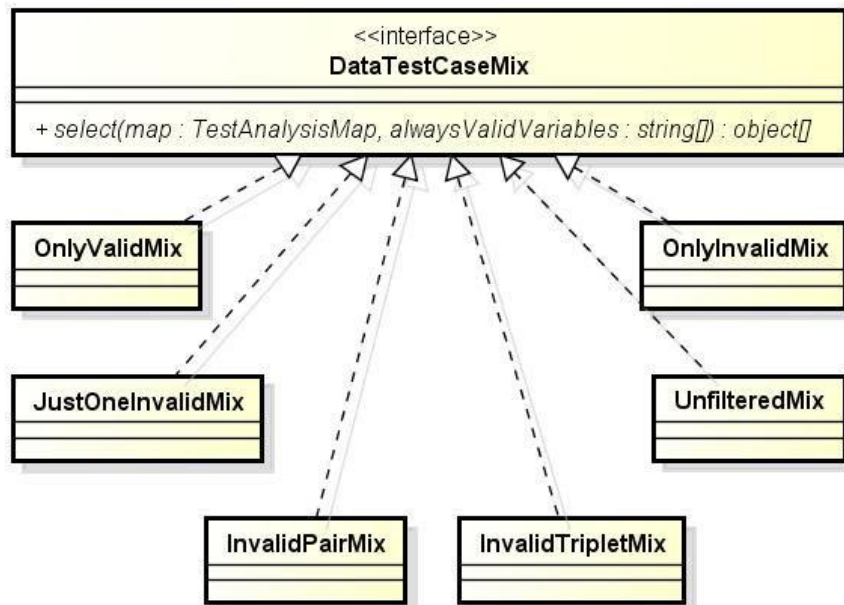


Figure A12 - Mix of Data Test Cases

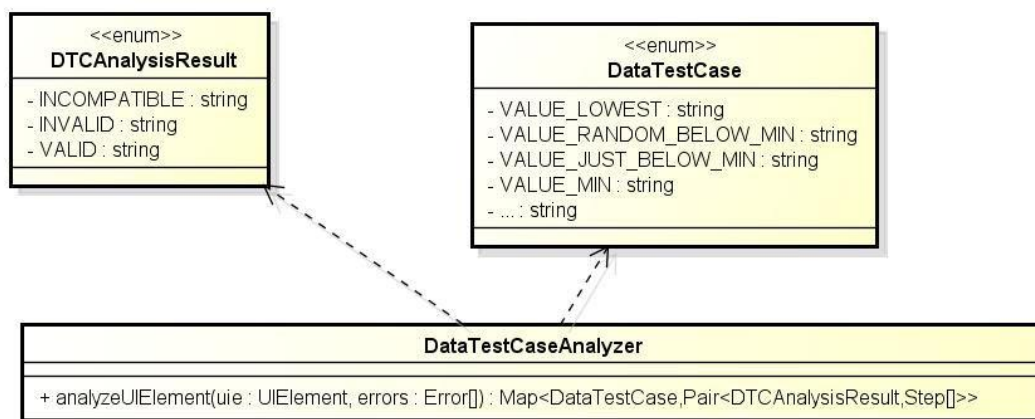


Figure A13 - Analyzer for Data Test Cases

We defined an analyzer (class `DataTestCaseAnalyzer`), presented in Figure A13, that evaluates the results (enumerated type `DTCAnalysisResult`) of every data test case (enumerated type `DataTestCase`) for a certain UI Element. It considers the UI Element properties and their values in this analysis – according to the approach described in section 7.3.5. In this way, we can establish the effect of inputs in UI Elements of a Test Scenario, and adjust its oracles accordingly. The method `analyzeUIElement` also returns the Oracle steps – retrieved from Otherwise sentences – associated with the result (`DTCAnalysisResult`). When a data test case is

considered valid or incompatible, no Oracle steps are returned (empty array). Otherwise, which means that the input produces a result considered invalid, the corresponding Otherwise steps are returned – whether they were specified.

Later, in the test case generation, when a data test case is considered *invalid* and *no Otherwise steps are specified* (i.e., the analyst did not specify the expected behavior for an invalid input in the corresponding UI Element), we are currently flagging the Test Case as “invalid”, which means that we expect it to fail. For example, whether a UI Element called “Price” has a minimum value defined as “0.01” (one cent), but it does not define what should happen when an invalid value is given (e.g., what happens if we inform “0.00”?), we are flagging the test case because its test scenario should not terminate successfully. We are considering to negate Variant’s postconditions (using NLP) in future versions, i.e., transforming Then sentences into their negated versions, to invert the original expectations – instead of letting the original oracle fails, as we currently do. For example, suppose that a Variant interacts with the UI Element “Price” mentioned above and also declares an oracle “Then I see the text “Saved.””. When we give an invalid price, such as “0.00”, we do not expect that the application shows “Saved” because we did not produce the needed conditions for that happens. Currently, we are flagging the Test Case with a tag “@fail”, to make the failure expectation clear. In future versions, we want to negate the oracle using NLP, to produce “Then I *do not* see the text “Saved.””, instead of flagging the test case.

A7. Test case generator

Figure A14 presents the structure of a Test Planner (class `TestPlanner`), that produces test plans (class `TestPlan`). A Test Plan is a *combination* of UI Elements, data test cases (enumerated type `DataTestCase`), and respective oracles (retrieved from Otherwise sentences). Later, the Test Case Generator (Figure A15) will apply a Test Plan into a Test Scenario to produce a Test Case. For producing test plans, the planner uses the strategies mentioned earlier (interfaces `CombinationStrategy` and `DataTestCaseMix`).

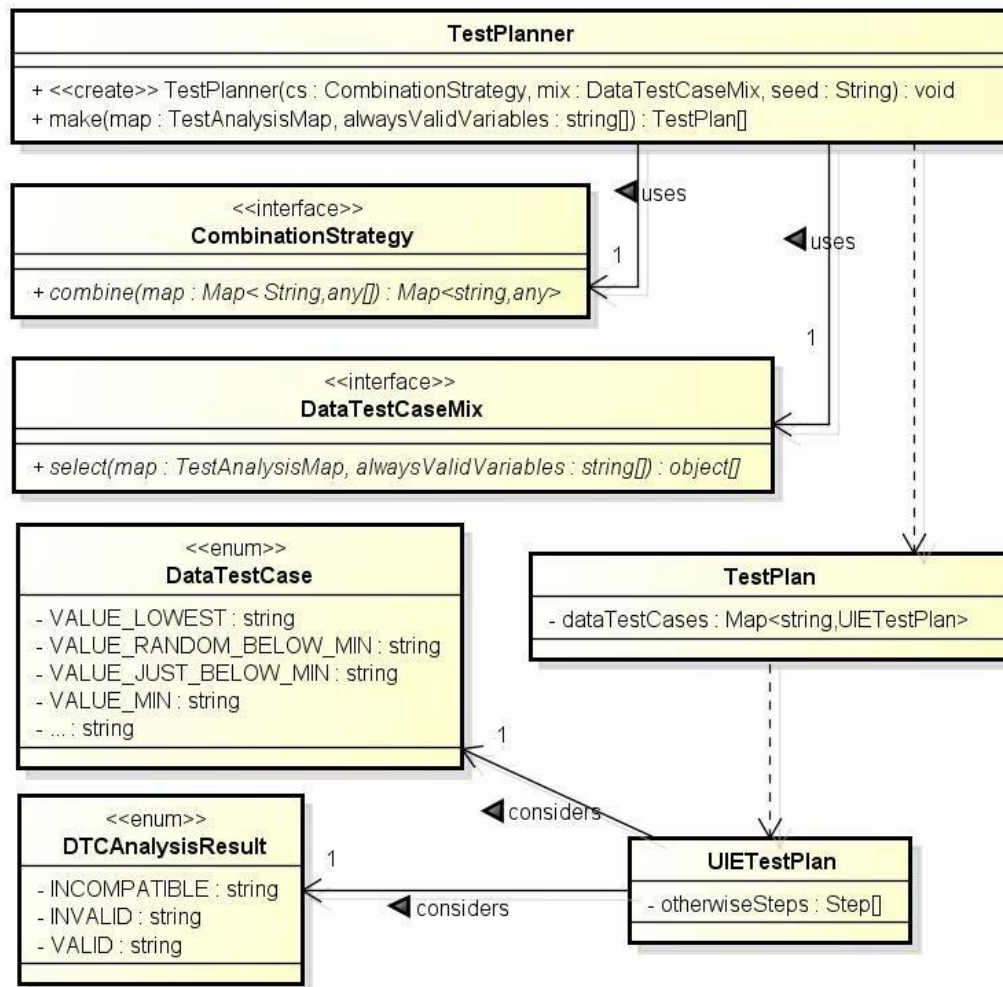


Figure A14 - Test Planner

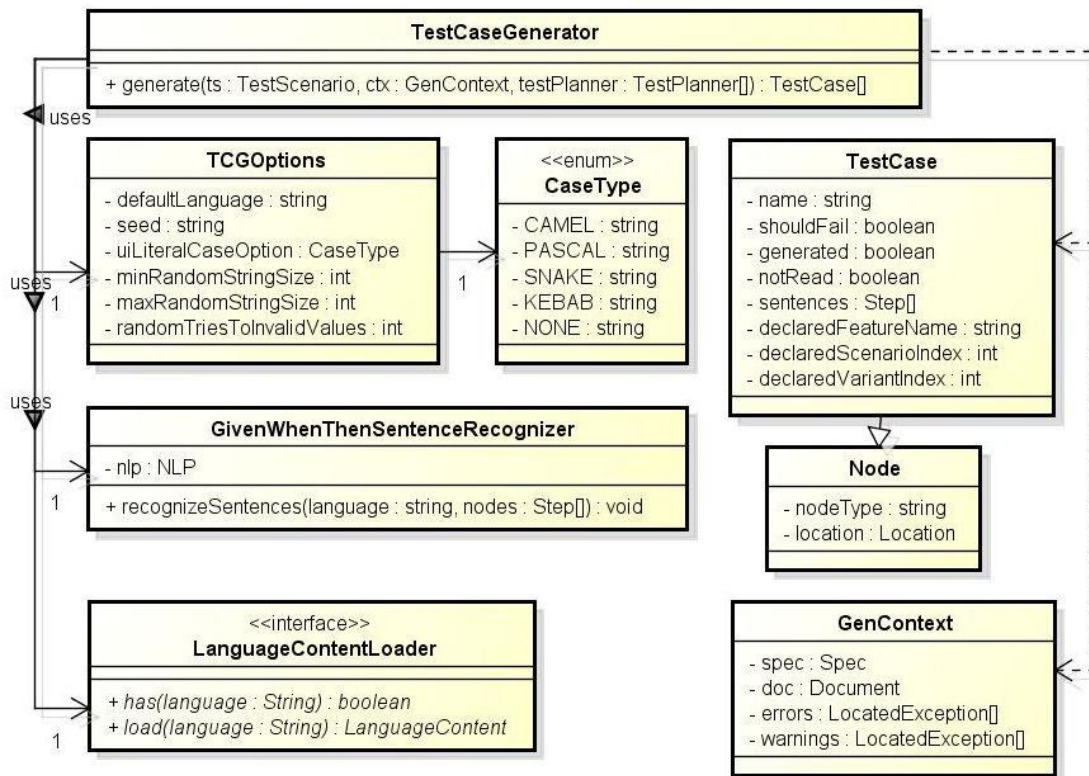


Figure A15 - Test Case Generator

Figure A15 shows the structure of the test case generator (class `TestCaseGenerator`). The generation considers parameters (class `TCGOptions`) such as the default language and the random seed, the generation context (class `GenContext`) – which includes the current document and the specification –, and a set of test planners (class `TestPlanner`). The process uses dictionaries – loaded by an implementation of `LanguageClassLoader` – to adjust Given-When-Then sentences' content. These sentences are analyzed again – with a `GivenWhenThenSentenceRecognizer` – to ensure that they have the right structure, *i.e.*, the right Intents and Entities, to that they can be transformed into Abstract Test Scripts later.

A8. Test script generator

Figure A16 shows the structure of an abstract test script (ATS) generator (class `AbstractTestScriptGenerator`). An abstract test script (class `AbstractTestScript`) contains all the needed information for producing test scripts, such as a feature, scenarios, and test cases (class `ATSTestCase`), and may have test events

(class ATSEvent). Both test cases and test events have commands (class ATSCommand) that can be converted into source code, later, with a plugin. Each of these commands (as well as any object possessed by an ATS) has the location of its respective sentence in the specification file. These locations provide traceability for the source code.

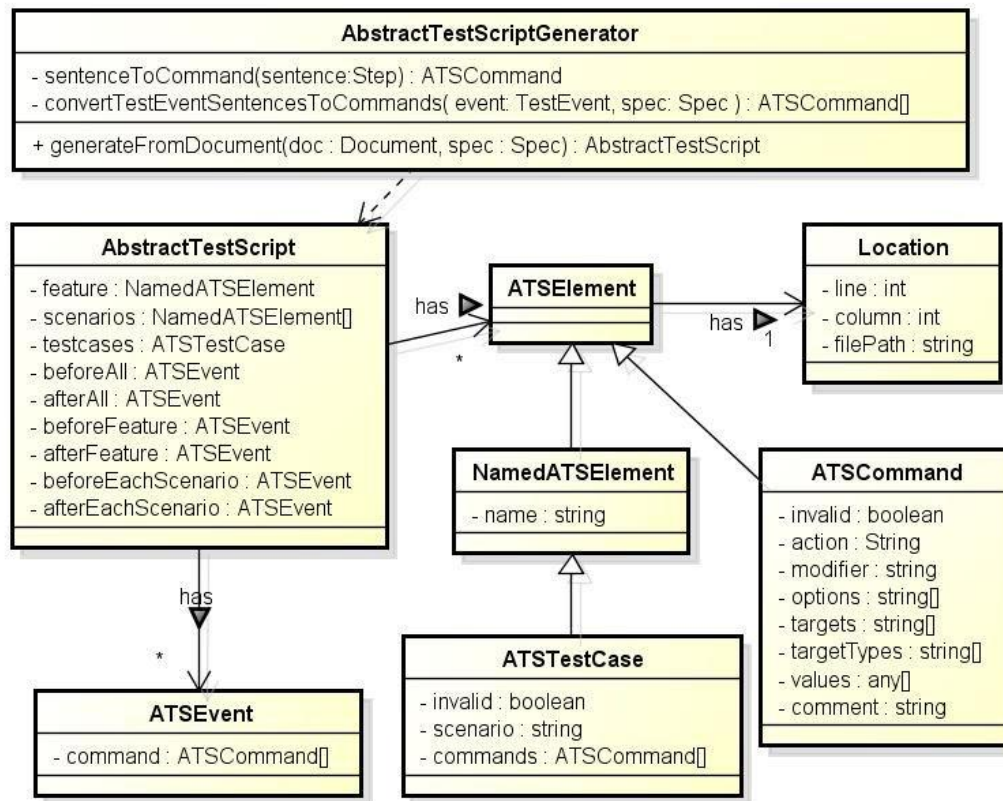


Figure A16 - Abstract Test Script Generation

Figure A17 presents the adopted plug-in-based architecture for test script generation. The interface **Plugin** can be implemented to generate source code for the desired testing framework. For example, currently there is a class named **CodeceptJS** that implements **Plugin** in order to generate test scripts for the framework **CodeceptJS**⁷³. The interface **Plugin** defines three methods:

- **generateCode**: transforms the given abstract test scripts (objects of the class **AbstractTestScript**) into source code, considering the given options (class **TestScriptGenerationOptions**);

⁷³ <https://github.com/Codeception/CodeceptJS>

- `executeCode`: executes the test scripts according to the given options (class `TestScriptExecutionOptions`);
- `convertReportFile`: read the execution results (class `TestScriptExecutionResult`) from a report file produced by the testing framework.

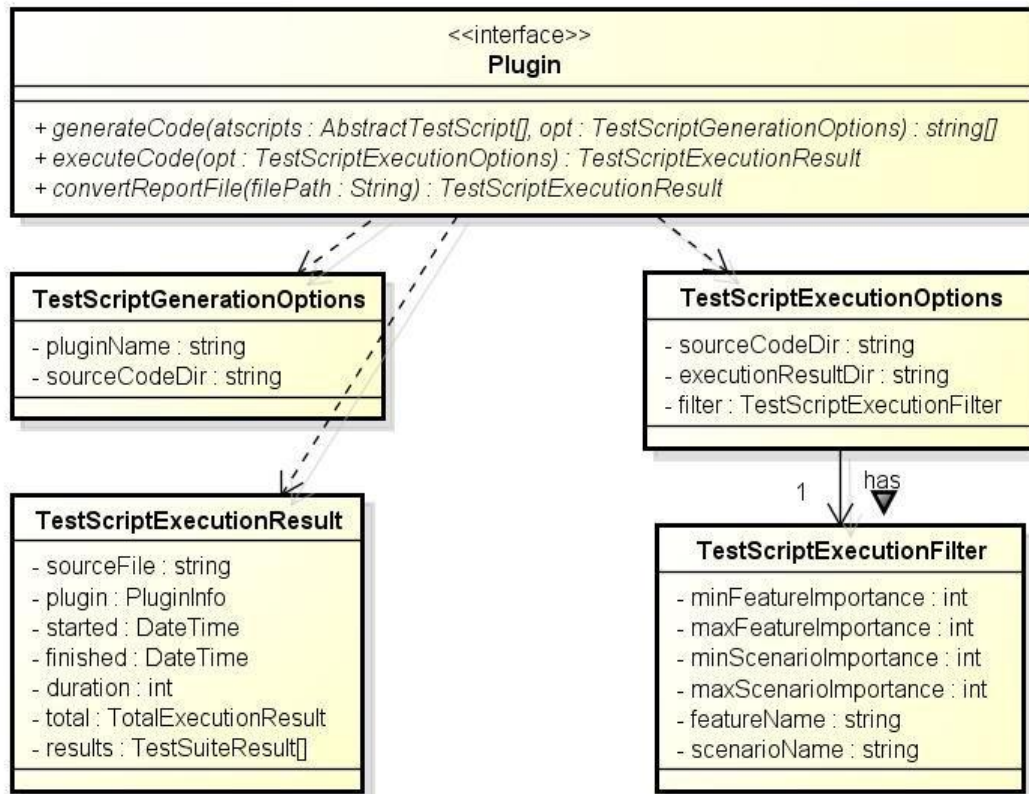


Figure A17 - Plug-in Architecture for Test Script Generation

A9. Test script executor and analyzer

The same plugin described in Figure A17 is responsible for executing test scripts and converting their results to the expected format. However, since the execution of test scripts may involve additional steps, such as starting a testing server that controls a browser (for web applications) or starting a mobile phone simulator (for mobile applications), we defined a structure (class `PluginData`) to define plugin data – see Figure A18. These data include a “serve” command that can be executed to setup a testing server, and commands to install and uninstall needed dependencies. Plugin data are currently stored as JSON files and loaded by a plugin finder (class `JsonBasedPluginFinder`). The “serve” command is only executed when a

user gives the parameter “--plugin-serve” to the tool, with the respective plugin name (*e.g.*, `concordia --plugin-serve codeceptjs`).

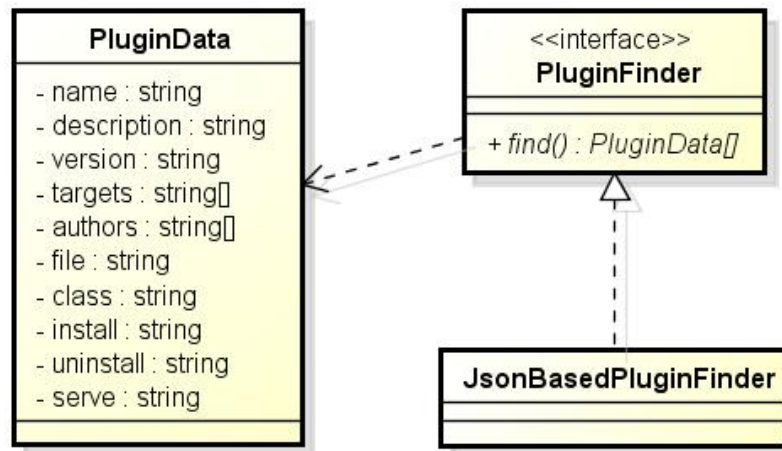


Figure A18 - Plug-in Data

The plugin data do not include a command to execute test scripts since there are (execution) options to be interpreted – the method `executeCode` from the class **Plugin** does that.

After using the plugin to read execution results from the corresponding testing framework, the tool analyzes these results to inform the user. For example, whether is expected that a certain test script fails (*i.e.*, when the test script is generated from a Test Case flagged with the tag `@fail`) and it really fails – which also means that the target testing framework does not offer a way to defining expectations of failure –, the tool can convert the failure into a success-like result (since the failure is expected). In this case, the result is reported as “adjusted” instead of as “successful”.

A10. Final Remarks

This appendix detailed the architecture produced to make the approach possible. It works like a compiler, transforming a high-level language (Concordia) into a low-level language (test scripts), using a large set of algorithms and natural language processing. The architecture can be extended to accommodate new algorithms (*e.g.*,

new combination approaches, new NLP algorithms) and new plugins for test script generation.

Appendix B – Concordia Grammar

The following listing presents the Concordia grammar in Backus-Naur Form (BNF) :

```
feature ::= white comment tags header background? feature_elements
comment?
header ::= (! (scenario_outline | scenario | background | variant_background) .)*
feature_elements ::= (scenario | scenario_outline)*
scenario ::= comment tags scenario_keyword space* lines_to_keyword
white steps
scenario_outline ::= comment tags scenario_outline_keyword space*
lines_to_keyword white steps testcase_sections white
background ::= comment background_keyword space* lines_to_keyword?
(eol+ | eof) steps
variant_background ::= comment variant_background_keyword space*
lines_to_keyword? (eol+ | eof) steps
tags ::= white (tag (space|eol)+)*
tag ::= '@' ([^@\\r\\n\\t ])+
comment ::= (comment_line white)*
comment_line ::= space* '#' line_to_eol
steps ::= step*
step ::= comment step_keyword keyword_space line_to_eol (eol+ | eof)
multiline_arg? white
testcase_sections ::= testcase*
testcase ::= comment space* testcase_keyword space* lines_to_keyword?
eol table white
multiline_arg ::= table | py_string
py_string ::= open_py_string (!close_py_string .)* close_py_string
open_py_string ::= space* '"""' space* eol
close_py_string ::= eol space* '"""' white
cell ::= [^\\r\\n|]+ '|'
row ::= space* '|' cell+ eol
table ::= row+
step_keyword ::= 'Given' | 'When' | 'Then' | 'And' | 'But'
```

```

import_keyword ::= 'Import'
testcase_keyword ::= 'Test Case:'
scenario_outline_keyword ::= 'Scenario Outline:'
scenario_keyword ::= 'Scenario:'
background_keyword ::= 'Background:'
variant_keyword ::= 'Variant:'
variant_background_keyword ::= 'Variant Background:'
property_line ::= '-' space* property connectors content
property ::= [^\r\n|]+
connectors ::= [^\r\n|]+
content ::= value | number
value ::= '"' [^\r\n|]+ '"'
number ::= [0-9]+(\.[0-9]*)?
table_keyword ::= 'Table:'
database_keyword ::= 'Database:'
constants_keyword ::= 'Constants:'
ui_element_keyword ::= 'UI Element:'
ui_element_step_keyword ::= 'Otherwise' | 'And' | 'But'
test_event ::= 'Before All' | 'After All' | 'Before Feature' | 'After
Feature' | 'Before Each Scenario' | 'After Each Scenario'
lines_to_keyword ::= (!eol space* reserved_words_and_symbols) .*
reserved_words_and_symbols ::= (step_keyword keyword_space) | (var-
iant_background_keyword keyword_space) | scenario_keyword | sce-
nario_outline_keyword | variant_keyword | variant_background_keyword |
table_keyword | database_keyword | constants_keyword | ui_element_key-
word | test_event | table | tag | comment_line | property_line
line_to_eol ::= (!eol) .*
space ::= ' ' | '\t'
eol ::= '\r'? '\n'
white ::= (space | eol)*
keyword_space ::= ' '

```

Appendix C – Static Checking

The following table presents the static checking performed by the proposed approach. It is a condensed list of verifications for the Concordia language. We merged some items to keep the list shorter.

#	Subgroup	Type ⁷⁴	Description
1	Any named declaration	E	Empty name
2	Any named declaration	E	Invalid name
3	Language	E	Language not available
4	Language	E	Just one declaration is supported
5	Language	E	Must be declared before an Import
6	Language/Import	E	Must be declared before a Feature
7	Import	E	Must have a file.
8	Import	E	Duplicated import
9	Import	E	File not found
10	Import	E	Imported file is a self-reference
11	Import	E	Cyclic reference
12	Feature	E	Feature has a duplicated name.
13	Scenario	E	Must be declared after a Feature
14	Scenario	E	Duplicated Scenario name.
15	Given/When/Then/And/Otherwise step	E	Must be declared for the following language constructions: ...

⁷⁴ Types are: E=Error, W=Warning

#	Subgroup	Type ⁷⁴	Description
16	Given/When/Then/And/Otherwise step	E	Must be declared after the following steps types:...
17	Given/When/Then/And/Otherwise step	E	Must have an owner.
18	Given/When/Then/And/Otherwise step	E	Referenced UI Element not found: ...
19	Given step of a Variant	E	A Given step cannot be declared after a step other than Given.
20	Given step of a Variant	E	Given steps with state must be declared before other Given steps.
21	Any NL sentence	W	Unrecognized entity
22	Any NL sentence	W	Unrecognized intent
23	Any NL sentence	W	Different entity recognized
24	Any NL sentence	W	Different intent recognized
25	Any NL sentence	E	Sentence expects different entities
26	Any NL sentence	E	Sentence expects an entity in different quantity
27	Any NL sentence	W	The sentence %s could not be validated due to an inexistent rule for property: ...
28	Any NL sentence	W	The property %s expects at least %d values, but it was informed %d'
29	Any NL sentence	W	The property %s expects at most %d values, but it was informed %d'

#	Subgroup	Type ⁷⁴	Description
30	Any NL sentence	W	The sentence %s could not be validated due to an inexistent rule for the target %s of the property %s
31	Any NL sentence	W	The property %s must be used with %s
32	UI Element	E	Must be declared after a Feature
33	UI Element	E	Duplicated local name.
34	UI Element	E	Duplicated global name.
35	UI Element	E	UI property must be declared after a Feature
36	UI Element	W	UI property not recognized
37	UI Element	E	Duplicated property
38	UI Element	E	Incompatible properties: ...
39	UI Element	E	Incompatible property operators: ...
40	UI Element	E	Referenced Constant not found: ...
41	UI Element	E	Referenced UI Element not found: ...
42	UI Element	E	Referenced Table not found: ...
43	UI Element	E	Referenced Database not found: ...
44	UI Element	E	Minimum value is greater than the maximum value
45	UI Element	E	Minimum length is greater than the maximum length
46	Tag	E	Invalid tag declaration
47	Tag	E	This tag must have a number.

#	Subgroup	Type ⁷⁴	Description
48	Tag	E	The tag content must be a number greater than zero.
49	Database	E	Duplicated database name
50	Database	E	Database property must be declared after a Database block
51	Database	W	Unrecognized database property
52	Database	W	Database property expects a value
53	Database	E	Could not connect to the declared database
54	Database	E	Error while disconnection from the database
55	Database	E	Database has no properties
56	Database	E	Database should have a type
57	Database	E	Database should have a property name or a property path
58	Query	E	Query cannot have both a reference to a Database and a reference to a Table.
59	Query	E	Query cannot have more than one Database reference.
60	Query	E	Error trying to process a database query.
61	Query	E	Query cannot have more than one Table reference.
62	Query	E	Query must have a Database reference or a Table reference.
63	Table	E	Duplicated Table name
64	Table	E	Invalid table row declaration

#	Subgroup	Type ⁷⁴	Description
65	Table	E	A table row must be declared after a Table declaration.
66	Table	E	Table must have at least two rows
67	Table	E	Error creating the in-memory table
68	Table	E	Error inserting declared data in the in-memory table
69	Constant	E	Duplicated constant name
70	Constant	E	Must be declared after a Constants block
71	Constant	E	Constant does not have a name
72	Constant	E	Constant does not have a value
73	Before All/After All/Before Feature/After Feature/Before Each Scenario/After Each Scenario	E	Event already declared
74	Before All/After All/Before Feature/After Feature/Before Each Scenario/After Each Scenario	E	Event must be declared after a feature
75	Feature/Background/Constants/any event	E	Already declared
76	Background	E	Must be declared after a feature
77	Background	E	Must be declared once
78	Background	E	Must be declared before a scenario
79	Variant	E	Duplicated Variant name
80	Variant	E	Required state is not produced by one of the imported Features.

#	Subgroup	Type ⁷⁴	Description
81	Variant	E	Preconditions refer to post-conditions produced by the owner Variant
82	Variant/Test Case	E	Must be declared after: ...
83	Variant/Test Case	W	Action not recognized
84	Test Case	E	No Imports or Feature declared before the Test Case.
85	Test Case	E	Imported document does not have a Feature.
86	Test Case	E	None of the imported documents have a Feature.
87	Test Case	E	Test case has no tag that refers to its Feature
88	Test Case	E	Tag refers to a non-existing Feature
89	Test Case	E	The referenced Feature does not have Scenarios
90	Test Case	E	Test Case has tag @variant but it does not have a tag @scenario. Please declare it.
91	Test Case	E	The index informed in @scenario is less than 1.
92	Test Case	E	The index informed in @scenario is greater than the number of scenarios.
93	Test Case	E	No Scenarios were found with the informed index.
94	Test Case	E	No Variants were found in the referenced Scenario.
95	Test Case	E	The index informed in @variant is less than 1.

#	Subgroup	Type ⁷⁴	Description
96	Test Case	E	The index informed in @variant is greater than the number of variants in the scenario.
97	Test Case Gen.	E	Error retrieving Test Scenarios from the Variant
98	Test Case Gen.	E	Error generating test case file
99	Test Case Gen.	W	Combination strategy not supported
100	Test Case Gen.	W	Data selection strategy not supported
101	Test Case Gen.	W	Variant selection strategy not supported:
102	Test Case Gen.	E	Could not generate a value for the following UI Element: ...
103	Test Case Gen.	E	Could not generate a value for the following UI Element property: ...
104	Test Case Gen.	E	Error trying to process the following database query: ...
105	Test Case Gen	E	A producer of the state was not found: ...
106	Test Case Gen.	E	Could not retrieve a value from the UI Element: ...
107	Test Case Gen.	E	Could not produce a UI Literal from the UI Element: ...