Simple Framework for Efficient Development of the Functional Requirement Verification-specific Language

Simple Framework for Efficient Development of the Functional Requirement Verification-Specific Language

Srdjan POPIC, Nikola TESLIC, Milan Z. BJELICA
Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
srdjan.popic@rt-rk.com

***Abstract*—This paper presents the framework for the creation of various domain-specific languages for verification of the functional requirements. When it comes to Requirement Engineering and the process of Validation and Verification of the requirements, there are plenty of tools for modeling, analyzing, and validating the requirements. It comes as a full-blown set of applications for validation of the requirements. But the set of the verification tools is either too complex or usable in a narrow domain. From the customers' point of view, there is a need for another independent requirement verification. This tool enables the creation of the custom verification in a way that allows users (either clients or developers) to verify requirements. It follows the IEEE guides, standards, and best practices to check all aspects of the software requirements that are neither implemented nor checked by the validation process: correctness, completeness, traceability, dependency, importance, and uniqueness. Tool implements design patterns specific to the verification process, thus enabling the faster implementation of the language. The concept can be used for development of the verification-specific language with any type of requirement representation, which will be shown by a few examples.**

***Index Terms*—computer languages, formal verification, formal languages, requirement engineering, programming environments.**

I. Introduction

The software development life cycle is a conceptual framework that defines the tasks that need to be executed at each phase. Its purpose is to deliver high-quality software. The list of the methodologies used for the software development lifecycle is very long. But, any given methodology of the development life cycle includes a set of processes called Verification and Validation [1]. The terms are used interchangeably, but there is a fine difference defined by the PMBOK (Project Management Body of Knowledge) guide [2]: validation - checks if a system meets the needs of the customer and verification - checks if the product complies with the requirements. This paper will be focused on the verification of the requirements.

Nuseibeh and Easterbrook [3] claim that requirements verification, consists of testing an actual implementation against the "precise specifications". This is the most usual way of verification, but it is not the only way to verify the requirements. Apart from verification by testing, the verification can be achieved by analysis of the model of a system or any of its parts. The verification by analysis is suitable when either system is not yet defined or available, or because it cannot be exercised directly due to cost, time, resources, or risk constraints [4]. It needs knowledge of the various, modeling tools and languages, that are complex to use, especially for the customers. Thus, the paper is focused on verification by testing. The process of verification is a process where all stakeholders are included. The supplier’s side has its ways to verify the requirements, usually as a part of the development cycle, with the tools widely known inside the supplier's company. The customers are not experienced with the tools and tend to have another independent verification. Customers also tend to verify by testing, since the complexity of the modeling. The framework presented in this paper gives the supplier a way to verify by testing its requirements independently.

The tool is the framework used to create an interpreter domain-specific language (DSL), precisely, verification-specific language (VSL) that will be able to verify by testing, the requirements' specifications of the software. To create the language, the user needs to map the language and requirement's specifics such as uniqueness, severity, traceability, dependency, recommended by the IEEE [5]. With the mappings defined, the interpreter of the source code can verify the requirements using implemented design patterns for constraints.

A. Paper Organization

This section describes the organization of the paper. The next chapter focuses on the basics needed for the framework: requirement specification and domain domain-specific languages. The third chapter looks for the related works either on verification languages or on testing languages, trying to find common ground and functionalities. The fourth chapter describes the solution: how it works, what are the basic terms and operations needed to implement the language based on the framework. It also explains four different implementation examples, that show all benefits of the framework-based language implementation. The fifth chapter reveals the result of all four implementation examples. The conclusion is at the end of the paper.

II. Fundamentals of the Verification-Based Language

In order to implement the verification-based language, we need to leverage all properties of the requirement specifications and domain-specific language as well.

This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia.

Digital Object Identifier 10.4316/AECE.2021.03002

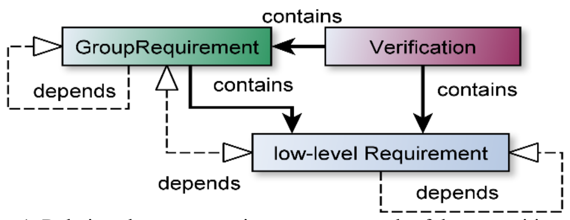


Figure 1. Relations between requirements as a result of decomposition

A. Requirement Specification

The requirement specification is a specification for a particular software product, program, or set of programs that perform certain functions in a specific environment. It may be written by one or more representatives of the supplier, one or more representatives of the customer, or by both [5]. Here is the list of best practices with reasoning why programming language is suitable for requirement specification:

- ***Correct*** – In order to ensure this, the requirement must be traceable in any direction [5];
- ***Unambiguous*** - Formal and understandable languages bridge the gap between customers and suppliers [6];
- ***Complete*** - All parts of the system must be covered. Programming language is easy to be checked for coverage;
- ***Consistent*** – no part of the requirement conflicts. Documenting requirements in natural language improves consistency [7]. *Preconditioning* is a process of exclusion/inclusion of requirements to be verified, which also makes the verification process consistent;
- ***Importance levels*** - Requirement verification tool needs implementation of *severity*;
- ***Verifiable*** – The process of the execution of the program written in a requirements-based language is verification;
- ***Modifiable*** – The requirement-based language must be able to implement *uniqueness* and *dependency*. Fig. 1 shows the dependency relations (dashed arrows) between the requirements in the verification process;
- ***Traceable*** - this is one of the most important properties and will be treated in a separate subsection below.

1) Requirement traceability

Requirement traceability is defined as "the ability to follow the life of a requirement in both a backward and forward direction", and it is a critical element of any rigorous software development process [8]. It comes even more into account with building safety-critical systems. Requirement traceability enables engineers to understand the ecosystem of the software primarily, the connections between different modules in the software system.

In literature, there are many definitions of different types of traceability, that can be enumerated and classified:

1. *Forward and Backward Traceability* [5]: Forward traceability is any link logically the next requirement. Backward traceability is the ability to trace back logically to all requirements;
2. *Horizontal and Vertical Traceability* [9]: Horizontal traceability is about tracing requirements at the same level of abstraction. Vertical traceability is the ability to follow the trace artifacts at different levels of abstraction.

To minimize naming confusion, there will be used different, names for different types of traceability. *Vertical traceability* is considered as the relation of **grouping** since more abstract requirements are divided into more concrete requirements. The more abstract requirement will be called a **group requirement**. Since the *horizontal traceability* links across the same level requirements only, it is considered as the **dependency relation on the same level** (either between group requirements or between single requirements). *Forward and backward traceability* is interpreted, as the relation of **dependency** between requirements of the different levels (requirement depends on group requirement).

2) Required functionalities of the requirement verification

As for the functionalities and properties of the verification process, there is a tangible consequential connection, between them and the best practices and IEEE recommendations for requirement specification. Table I displays the relations. Note, that the last three are properties of the language.

TABLE I. FUNCTIONALITIES AND PROPERTIES NEEDED FOR THE REQUIREMENT VERIFICATION PROCESS AND THE RECOMMENDATIONS THEY ARE ENABLING

Functionality	Recommended Characteristic
Unique indexing	Traceable, Correct and Modifiable
Dependency relation	Consistent, Modifiable and Horizontally Traceable
Circle dependency check	Consistent and Correct
Grouping relation	Vertically Traceable
Preconditioning	Consistent
Severity	Importance levels
Pre- and post- tasks	Complete and Traceable
Syntax check	Complete, Consistent and Correct
Well-structured language	Unambiguous and Consistent
Ability to extend	Modifiable, Consistent

B. Domain-Specific Language (DSL)

From DSL's point of view to build requirement verification specific language, it needs to support the specifics of the verification process:

- **recommended properties**: Uniqueness, Severity, Traceability (mandatory for safety-critical software), and Dependency;
- **functional constraints**: requirement grouping, check on cyclic dependency and preconditioning.

Apart from this, the characteristics of the language, such as syntax check and ability to extend, additionally strengthen the reasoning for the usage of the domain-specific language for the verification of the requirement specification.

DSL uses familiar concepts and rules from the specific domain such as the language elements and constructs. The domains may vary, but there is one common thing: narrowing the focus of the domain. The narrow focus makes things easier to define, understand, and most importantly execute.

The main aspect of the DSL implementation and usage in the development process is **synchronization**. Figure 2 shows the main processes in the development life cycle.

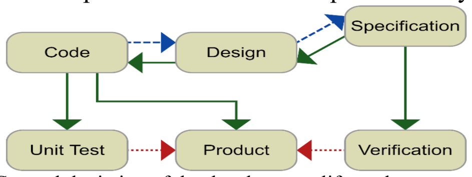


Figure 2. General depiction of the development life cycle

The green arrow shows the primary flow: requirement specification – design – coding. The coding is then used to build the product, but also to implement the testing of the product. At the same time, requirement specification is used for the verification of the same product. This influence is the most important reason for DSL usage. The red arrows show the testing influence. The feedback steps (depicted with a blue arrow) are used to correct the process above. The coding process changes the design, which changes the specification. This shows clearly the need for synchronization: In case there is no direct and automatic influence of the requirement specification on verification, the verification process won't be useful for the verification of the product. When the same DSL is used for the specification and the verification of the requirement, **the synchronization is achieved automatically**.

Finally, the usage of the DSL on the specification level clears the ambiguities.

There is a wide range of DSL usage in many areas of industry, mainly created in-house, without any external support [10], which shows that inside, expert knowledge is the most important in the creation of the DSL.

1) *Flavors of DSLs*

The Domain-specific languages by their representation can be **textual**, such as XML Path Language (XPath), or **graphical** such as UML (Unified Modeling Language). In a sense of the technique used, Fowler [11] groups DSLs into three strands: *external*, *internal*, and *language workbenches*. The third way of separating the DSLs is: *Compiler* (also known as Generator) and *Interpreter* (or Engine). A generator takes the source code program and transforms it into a lower-level object that can run directly on the target platform. The interpreter is a program that reads and interprets the source code.

2) *The expenses and benefits*

A good DSL can describe a system behavior's critical parts in ways a domain expert can understand [11]. But this comes with the price since the engagement of the expert is not cheap. The costs for domain-specific modeling will be less than the costs of general-purpose modeling [12]. Initial costs of the DSL are higher due to the engagement of domain experts and experienced developers at the start of the project, but afterward, it is much cheaper. The more products are built based on the DSL, the more likely is the return of the investments.

Except for the lesser costs, there are other benefits. The larger the number of application developers, the greater are the benefits of having a DSL solution [12]. Also, less experienced developers found themselves able to effectively develop application features [12] and more experienced perceives the DSL approach easier. This improves the communication between domain experts and all other stakeholders in the process: developers [11-12], business leaders, and financial [12-13].

With the VSL Framework, developing the language appears to be even easier, since all common concepts of the verification process are already implemented.

III. Related Works

The paper describes the tool that is the framework for implementing verification-specific languages and executing the verification of the requirements. From that point of view, there are no similar works that can be related. The framework intends to support various verification processes, and there are plenty of tools proposed for requirement verification processes. The main reason for introducing the framework instead of a single VSL (or any other kind of verification tool) is the framework's ability to support any terminology given by the requirements. This terminology is usually changed from project to project, which makes it impossible to reuse the same verification process in another environment (development life cycle). The subsections below examine the works where the language that is used for requirement description is used for verification too.

A. *User Requirement Notation*

User Requirement Notation (URN) [14] is a visual modeling language that aims to support the elicitation, analysis, specification, and validation of requirements. It defines scenarios and goals, which makes it available for requirement verification too. Amyot, Logrippo, and Weiss [15] recognize three different approaches where URN is used for verification purposes: testing patterns, scenario definitions, and transformations to formal specifications.

These three concepts of requirement analysis that can be used for test generation are considered as verification. But all three concepts have their downfalls, with the note that the scenario definitions appear to be the best of all three.

B. *RSLingo*

RSLingo [16] is a requirement-specific language for improving the quality of requirements' specifications. The process mimics the human process of reading and understanding of requirement text. It is the language used for extracting the meaning and the glossary of the requirements focused on the better specification of the requirement and a better understanding of all stakeholders.

From the verification point of view, it can verify some of the requirements as the side-effect, mostly in a process of the assessment of requirement quality criteria. This is not the tool that can cover full requirement verification.

C. *Alneelain Language*

Alneelain [17] is a compiler requirement-specific language based on axiomatic specification. It uses a recursive notation that can be translated almost directly into recursive programs. The language is based on the model proposed by Mili and Tchier [18], where the requirement specification must follow two conditions: *formality* - achieved by mathematical notation and *abstraction* - the description of what requirements the software product must satisfy, not how to satisfy them.

The mathematical notation makes this language very difficult to use for the non-math customers, who would need to use it for independent verification. Here is how the stack is specified with this mathematical notation:

$X=\{init, pop, top, size, empty\} \cup \{push\} \otimes itemtype$

D. Requirement Verification with Promela

Hong and Ming [19] proposed requirement verification with a semi-formal ontology-driven domain-specific requirements specification language. It is a graphical, external, compiler DSL that generates Promela source code that can be executed using SPIN (Simple Promela Interpreter), to verify the requirements.

It seems like a suitable language for the customer, since it is a graphical tool, and the ontology that is built in a process, may be the best way to communicate with the customer, and with the machine too. The proposed solution is still in the design stage and has not been tested by the prototype, but shows a good pathway for verification. However, the specific properties of the verification process such as the level of importance (severity) have not been built.

E. Requirement Modeling Based on the Restricted Natural Language (RM-RNL)

RM-RNL [9] is the language based on restricted natural language and architecture analysis and design language (AADL) [20] used for requirement specification on the safety-critical software (SCS). It is a model-based language, where the models are used to analyze the system, create, and communicate requirements. It enforces the requirement traceability of all types.

From the verification perspective, this language provides an integrated verification environment for its models using existing verification tools, such as Timed Abstract State Machine (TASM) [21], UPPAAL [22]. The verification of the highest group requirements is based on another tool called AGREE [23]. It supports almost all important features of the verification process, except for the cyclic dependency. Also, a user needs to learn one modeling language and master one of the verification tools.

F. Controlled Natural Languages

Controlled Natural Languages (CNL) are part of natural languages with restricted grammar, glossary, and dictionaries. The goal is to reduce both ambiguity and complexity. Traditionally, CNL falls into two major categories: those that improve the readability for human readers, and those that improve the computational processing of a text.

The biggest flaw of the CNL is that they are easy to read, yet hard to write. Another downside is the fact that usage of the natural language in the verification process neither implements the properties recommended by the IEEE nor constraints of the requirement specification.

A number of the CNLs are used in a process of verification. At first, these languages are used only for the clear writing of the requirements, but, some of them grow into the verification tool, either extended or in collaboration with additional tools.

1) Attempto controlled english (ACE)

ACE is a subset of English that can be unambiguously translated into first-order logic. The parser generates a syntax tree which can be translated into other logic languages such as TPTP (Thousands of Problems for Theorem Provers) [24], OWL 2 Web Ontology Language [25], and SWRL (Semantic Web Rule Language) [26].

The problem is the fact that the logic extracted from the requirement can be undecided, hence infinite. Efficiency and scaling-up behavior have not yet been systematically investigated [27].

2) Computer-processable language (CPL)

CPL is designed with a naturalistic approach with the ability to resolve any ambiguity in sentences and produce correct implementations [28]. CPL can convert textual requirements written in a given manner into the programming language and used for testing purposes. Unfortunately, it is unable to follow the recommendations from the IEEE, with the exclusion of the precondition design pattern using questions. It requires a degree of training to be able to construct the requirements.

3) Automating test automation (ATA)

ATA is a tool that enables the execution of test cases written in natural language and creates a representation suitable for automated execution [29]. It is suitable for the testers who are, generally, not programmers. The downside is that the descriptions could contain ambiguity.

G. Existing DSL Testing Tools

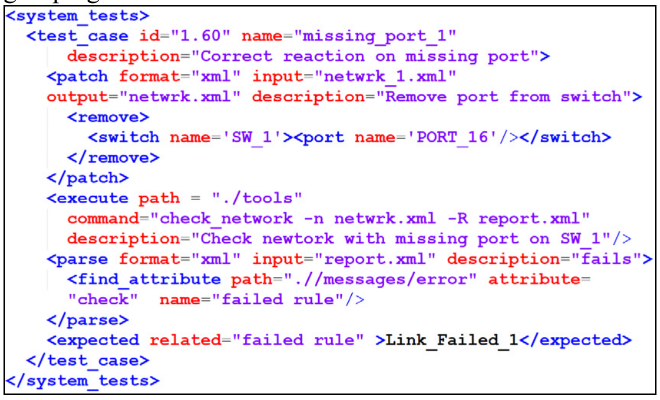
Almost every testing tool can be used for verification, it is suitable to consider all Testing-specific languages. Some of the DSLs (Photon [30], Gatling, Canopus [31], CONCEPTUAL [32]) are focused on performance testing, which is not the focus of this DSL. Other DSLs (such as ATAP [33], TTCN-3[34], Simulink [35]) need to model the system to test it. This means that developers must have a strong technical, but a business background too.

Last, but not least, none of them offers a traceability mechanism, so the requirement can trace the test case.

H. Previous Research - Testing DSL in V-Model Life Cycle

The paper that preceded this research [36] is a simple interpreter testing-specific language (TSL) that supports traceability. Fig. 3 shows the source code example of one test case written in this language.

The downsides that triggered this research are: plenty of new terms (executor, patcher, parser ...), knowledge of the XML language, missing spell checking, cyclic check, and grouping.



```
<system_tests>
  <test_case id="1.60" name="missing_port_1"
      description="Correct reaction on missing port">
    <patch format="xml" input="netwrk_1.xml"
    output="netwrk.xml" description="Remove port from switch">
      <remove>
        <switch name='SW_1'><port name='PORT_16'/></switch>
      </remove>
    </patch>
    <execute path = "./tools"
      command="check_network -n netwrk.xml -R report.xml"
      description="Check newtork with missing port on SW_1"/>
    <parse format="xml" input="report.xml" description="fails">
      <find_attribute path=".//messages/error" attribute=
      "check"  name="failed rule"/>
    </parse>
    <expected related="failed rule" >Link_Failed_1</expected>
  </test_case>
</system_tests>
```

Figure 3. An example of the test case

I. Overview of the Related Works

There are four most important aspects of the VSL:

- The type of the verification: either *verification by analysis* or *verification by testing*. The advance is on the testing since the analysis depends on a model that can have indefinite states;
- The set of properties that needs to be implemented in the language, recommended by the IEEE: Uniqueness, Severity, Traceability, and Dependency;
- The complexity of the language answers the question: *How hard is it to learn and create verification?*
- The set of design patterns that need to be implemented within the verification language: checking the cyclic of the dependencies, checking the prerequisites of the requirement (preconditions), and check the grouping relation between the requirements.

Table II shows the feature overview of all of the above tools/languages. Since the test-specific languages are mostly aligned with the properties of the verification process, they are grouped in one row. Properties recommended by th IEEE are **U**niqueness, **S**everity, **T**raceability and **D**ependency. Constraints presented in the table are: **C**yclic check, **G**rouping and **P**reconditioning.

TABLE II. FEATURE OVERVIEW OF THE TOOLS/LANGUAGES USED FOR REQUIREMENT VERIFICATION

Verification Tools	**Features**			
	Verification Type	***Recommended by IEEE***	***Complexity***	***Constraints***
URN	Analysis	U S D T	High	No
RSLingo	Testing	U D T	Medium	No
Alneelain	Testing	D	High	No
Promela	Analysis	U S D	Medium	P
RM-RNL	Analysis	U S D T	High	G P
ACE	Analysis	None	Medium	No
CPLs	Testing	None	Medium	P
ATA	Testing	None	Low	C
Testing tools	Testing	S D	Medium	P
Previous research	Testing	U T	Medium	P

IV. THE SOLUTION

The solution must be understandable, without ambiguity for both sides: customers and developers, and executable. The language elements (domain concepts and rules) used for specifying the requirements, must have all necessary properties to be used for the verification process. Any language based on the framework supports all the properties and functionalities noted in the Requirement Specification chapter, Table I.

The solution, proposed here, is the engine-based framework developed in Python that offers its API for mapping all mandatory properties of the VSL. It supports the development of the textual, interpreter (either internal or external) languages and the execution of the source code. The parts of the syntax checks such as uniqueness, indexing, grouping, and dependency are implemented in the engine.

The engine of the developed VSL wraps the Framework engine by implementing only the functionalities specific to the given VSL representation. The loading and parsing are a concern of the newly developed engine and the reporting and output, on the other side, are primarily a concern of the framework engine.

```
arr[9] # 10th element of an array
struct.thePart # the part of the structure
struct.thePart.arr[9]
struct.thePart[name="partName"].arr[9] # filtering
struct.[value<5, value>3].thePart # filtering
```

Figure 4. Default variable accessing examples

A. Variables

The variable syntax has its default (framework's) representation that can be used, and it can be redefined. If variable representation is redefined, some other default functionalities, such as prerequisites, cannot be used off the shelf. Variable values are lazy-loaded.

1) Default variable notations

The variables are defined by the rules given by the developed language, but the way to access to variable's data is already defaulted by the framework. The access to a simple variable is by the variable name.

But in the case of more complex structures such as the sets, lists, structures, and maps, the access can be divided into two groups: access by index (lists, sets, tuples...) or access by property name (structures and maps). Figure 4 shows some examples of how to access variable values.

2) Requirement's severities

Severity defines the level of requirement's effect on system functionality and quality. Every requirement has a severity as the mandatory property, but what are the levels of the severity? Default values are CRITICAL, MAJOR, MEDIUM, and LOW. Every severity value has its numeric representation used in the process of final verification result calculation. If these numeric values are not defined, they are distributed evenly from 0 to 100. In the default case, LOW severity has value 0, MEDIUM 33.33, MAJOR 66.66, and CRITICAL 100. The values can be redefined in any way: Change the name and the number of severities and change its numeric value. When the default set of severities is used, the default severity for every Requirement is MAJOR.

3) Verification results

On the other side, every requirement verification has a result. Usually, there is PASS, FAIL and NOT EXECUTED (for the situations when it is not possible to verify the requirement.). But sometimes, there is a need for the fine granularity of the verification results, or there is a need for distinction between verification not executed due to prerequisites fail and verification not executed due to dependent requirement fails. The verification result of one requirement concerns the whole verification process due to dependency relations between requirements. This means that some other, specific set of the results must be mapped into three default result values. The final constraint in a redefinition of the verification results is that at least two results must exist: one mapped into PASS and another mapped into FAIL. Along with the (re)definition of the results, two additional situations (called ***trimming situations***) need to be mapped:

- to the precondition fails;
- the case when a requirement is not verified due to dependencies.

B. Requirement Template

The requirement template is the "connection bridge" between the newly developed language and the framework. It enables the VLS representation loader to load all mandatory data needed from the verification process, such as: *id*, *severity*, *group* it belongs, *prerequisites* (in a form of reference to the functionality, since they can be redefined), the list of *dependencies*, along with some optional data such as the name, the description of the requirement and the list of tags (this list can be used to represent additional relations between the requirements, other than grouping and dependency). The requirement template has basic functionalities that are partially or fully implemented: `injectData` is the function that loads the concrete values of the variables needed in a process of the verification of concrete requirement, `syntaxCheck` checks what framework can check regarding the requirement (*id* is unique, *group* exists, *dependencies* exist, *severity* is from the set of (re)defined list of severities...), `verify` points to the verification function defined in VSL, and `preVerify` and `postVerify` functions enable the pre- and post-tasks noted in Table I, thus support Completeness and Traceability of the requirements. These two last functions are considered to be similar to *setUp* and *tearDown* functionalities in the testing process.

C. The Execution Responsibilities

All parts of the verification process exist in the Framework engine, but some parts are just placeholders or abstract functions, such as parsing of the source code, loading the context/"universe of objects" (the set of the variables and functions that will be re/used in the execution). Other parts of the process, such as syntax checking are partially defined and executed in the framework, and partially defined and executed in the newly developed engine. The more detailed division of the process responsibilities between developed and framework engines is shown in Figure 5.

1) Loading and syntax check

The execution process begins with the loading and parsing of the source code, which is specific for every representation of the VSL. This is where the abstract syntax tree is created. The next step is the loading of the "universe of the objects" where all requirements are loaded using the requirement template and function loader (will be explained in Extension subchapter below) from the framework. Here all mandatory properties of the requirement must be filled, but the functionality of the verification still lies in the VSL parser. Regarding preconditions for the verification, the plain Python syntax can be used, but if it is not used then it needs to be parsed with the VSL parser. Figure 5 shows the precondition as the responsibility of the newly developed engine, which is not always the case.

The syntax check function is distributed through the parsing and loading process, but there is a part of the syntax check on the given language that is executed after the context has been loaded. The checking of the syntax is extended with framework syntax checking that is consisted of various checks, such as the uniqueness of the indexes (and other things checked within every requirement template), cyclic checks of the grouping, and dependency as the relations between the existing requirements.

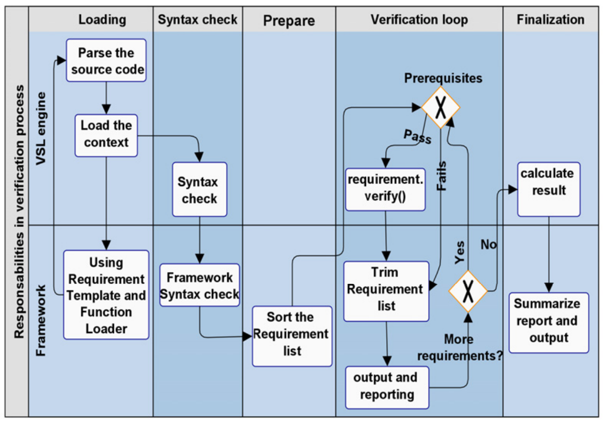


Figure 5. Responsibility sharing in the execution process

2) Sorting the requirements

In the process of the framework's syntax check, the stage is set for the sorting of the requirements. That in case if there is no cyclic infringement. Here we must say, that the order in which the requirements are specified in the source file is not relevant for their execution. That sounds like a sane suggestion, due to the dependency relations: to verify the requirement, all its dependent requirements must be verified first. The Fig. 6 shows one example the dependency execution tree, where the requirements are placed in the levels, based on the highest level of the dependent requirement.

The requirements without the dependencies are on level 0. Level 1 contains the requirements that have as much as possible dependencies, but all those dependent requirements do not have any dependency, and so on ... . First, all requirements from level 0 are verified, then level 1, ... Another rule of the dependency execution tree is that the most left requirement in the tree's level is the requirement with the smallest ID. This enables that every time the same verification process is executed, the order of the requirement verification will be the same. The requirement order for the example from Figure 6 is: VC1, VC3, VC4, VC8 (level 0), VC2, VC6, VC9 (level 1), VC5, VC10 (level 2), VC7 (level 3).

3) Verification loop

The sorted list of the requirements is iterated through the verification loop that executes functions from the framework together with the functions defined in a new programming language, interchangeably. If the requirement contains the prerequisites, they will be checked, and if prerequisites pass, the requirement's function `verify` will be executed (wrapped with `preVerify` and `postVerify` functions).

After verification, the flow joins with the failing prerequisites on the *trimming process*. The *Trimming Process* does nothing in case prerequisites and verification passes, but if either of the processes fails, the trimming process removes all requirements that depend on the current one from the requirement list. If the list of possible results has been redefined, the trimming will be considered only in cases when the verification results are the one that is mapped to FAIL or NOT EXECUTED (since all redefined results must be mapped into default ones).

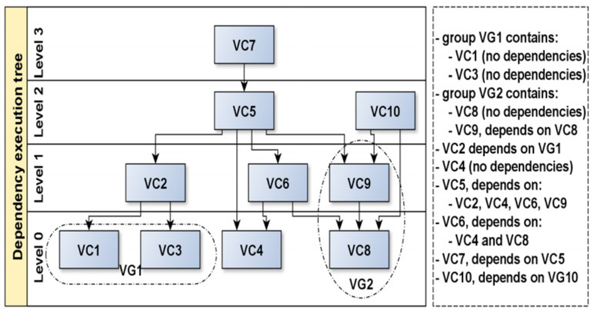


Figure 6. Example of the sorting

All trimmed requirements will be marked with one of the results defined as trimming situation results (mentioned in sub-chapter Verification Results). After trimming, all the results (verified requirement, and possibly skipped requirement) will be processed with the framework's output and reporting functions. After the output and report data process, the loop continues iterating on the rest of the requirement list.

4) *Finalization*

It calculates the overall result of the verification process and summarizes it in the report and output. There is a default calculation of overall results implemented in Framework. The default overall result calculation (shown in Figure 7) assigns to every verification the weighting factor equal to the severity value of the given requirement and then summarizes all weighting factors for failed requirement verifications. If this sum is greater than zero whole verification process is considered as failed, otherwise, if at least one requirement verification passes, the complete process pass.

It means that the failed requirement with severity LOW would not trigger the overall failing result. Also, if there is a need for different overall result calculations, the process needs to be overwritten. Another part of the finalization is strictly the responsibility of the framework. It finalizes the report and standard output based on the calculations.

D. *Extension*

To support the requirements and the language to be Modifiable and Consistent (Table I), the framework should be able to extend itself. The extension is based on Python's way to extend the code.

Similar to the Requirement Template, which is used as a communication bridge between a developed engine and the framework engine, Framework uses the *Function template* shown in Figure 8. The loading process scans **lib** folder (including the subfolders) for python files. It creates the function and parameters set for every function in every python file. The `name` is the name of the function in the python file. The `module` will be a file path relative from **lib** folder, with the dot as a folder separator. In a source code parsing process, the parser searches for the function in the function bag based on the module and the name of the function. If the function is found, it uses it for late syntax check and verification processes.

E. *Implemented Examples*

1) *VSL for previous research based on framework engine*

This is the DSL language used for testing presented in previous research [36]. The language representation, based on XML language, has been developed by another developer with comparable developing skills. To implement a framework engine in the solution, the new version of the DSL has grouping, dependency, and severity.

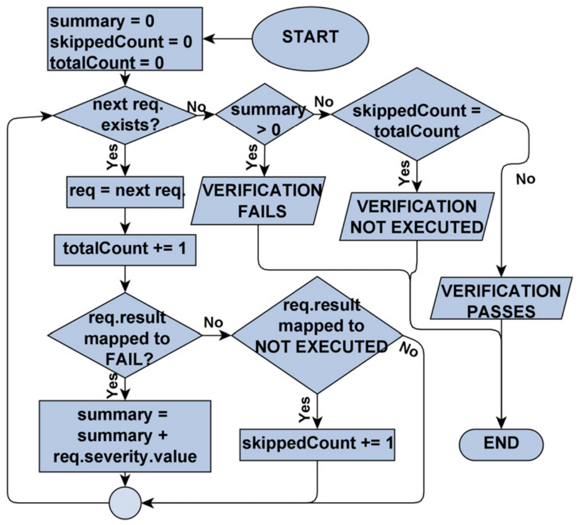


Figure 7. The default overall result calculation

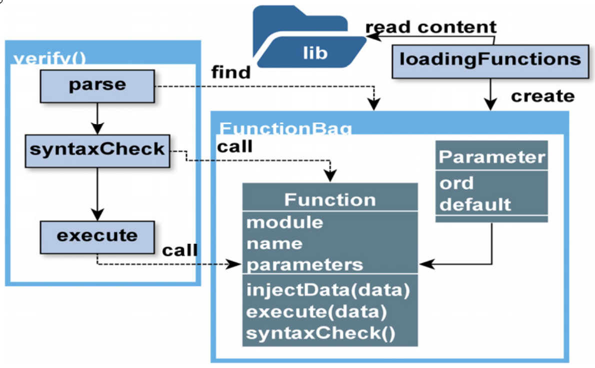


Figure 8. Framework's function bag

Apart from the extra time needed for understanding the framework platform, the developer spent the same amount of time to reason the requirements on what to develop. Also, the user's documentation for both developments appeared to have similar clarity and time needed for its creation. The time for reasoning the requirements of what to develop and time for the creation of the user's documentation are fairly equal. The developed DSL aligns with the framework default properties and functionalities, default severities and results were used and none of the functionalities has been redefined.

2) *Enhancement of the existing verification process*

The existing verification process has been fully developed as the set of the Python functions, grouped in modules and packages. The verification process has not been aligned with the requirement verification properties. The so-called *function-requirements* just have been called in a statically defined order. The extended version of the verification language is internal Python-based VSL. Figure 9 shows the example of the source code of the new VSL. Grouping is implemented as a module or package, containing the *function-requirements*. The module variable named `id` represents the group id and the module variable `depends` represents the list of requirement/group ids that the group is depending on. The dependencies of the *function-requirement* are defined by the decorator `@dependency`.

```
from req_lib import *
id = group_id # group id
depends = []  # the ids group depends on
@requirement(1, "ERROR")
def req_1(table): # requirement id=1, severity=ERROR
    # requirement verification body
    if verification_pass:
        return True
    return False
@requirement(2, "WARNING")
@dependency(3, 1)
def req_2(data1, data2):# id=2, sev.=WARNING, deps.= 3,1
    # requirement verification body
    if verification_pass:
        return True
    return False
```

Figure 9. The extract of the internal Pythonic VSL

The `id` and the `severity` of the simple requirement are defined by the decorator `@requirement`. The essential parts of the verification functionalities stay the same.

3) *Extending the specification language*

Since the framework enforces all properties of the specification language, it can be used to extend current specification languages. In this subsection, the focus is on the requirement specification language used to unambiguously define the requirements and to divide them into smaller, more precise requirements. This specification language is represented with YAML markup language and then extended using the framework to become executable as the verification language. One of the top-level requirements is: ***The configurable application parameters will be stored in the configuration file, named `appName.cfg`. The configuration file will be stored in a user's home subfolder named `appName`. If the configuration file or folder `appName` does not exist, the next run of the application will create one with default values. The default values of the configurable parameters are:***

- ***parameter1 = value1***
- ***parameter2 = value2***

This requirement was stated as a group requirement with the id="30-A". Figure 10 does not show the group requirement, but it shows some of the requirements derived from it. Parameter `id` is used as the unique identification of the requirement, `desc` is used for the description - not in the interest of the verification engine. The list of parameters defined in `dep` is either requirement ids or group ids.

The difference is made in the `id` naming standard: if the id ends with `-A` it is a group requirement. Based on the snippet from Figure 10, requirement `301-1` depends on `300-1` and together with `300-1` is grouped inside the higher-level requirement `30-A`. The syntax check for the naming standard of the `id` already exists within the current requirement specification language. The `setup` is the preparing process (aligned with `preVerify()`). It is treated as the natural language statement. For example, the statement:

`folder ~/appName does not exist`

is translated into the command:

`rm ~/appName`.

The parameter `function` is plain OS executable command. The parameter `outcome`, treated as the natural language will be translated into the set of commands, resulting in TRUE or FALSE. The parameter `output` will parse and read either output file or standard output or error pipeline (based on the `type` parameter). Verification result is logical AND between all outcomes and outputs.

```
id: "300-1"
  desc: "If folder userHome/appName does not exist, appName
  start will create it."
  dep: "30-A"
  setup: "~/appName does not exist"
  function: "appName"
  outcome: "exist ~/appName"
  output: "folder created"
    type: "file"
    name: "log/%.log"
id: "301-1"
  desc: "If userHome/appName/appName.cfg does not exist,
  next appName start will create it with default values."
  dep:
    - "30-A"
    - "300-1"
  setup: "~/appName/appName.cfg does not exist"
  function: "appName"
  outcome: "exist ~/appName/appName.cfg"
  output:
    - "parameter1:value1"
    - "parameter2:value2"
    type: "file"
    name: "~/appName/appName.cfg"
```

Figure 10. The snippet of the requirement specification file

The preconditions were not used here and the severities were used as-is. The system used default results and overall result calculation.

4) *Generated data verification*

The last implementation has not been used to verify the application but to verify the data produced by the artificial data generator. Artificial data can be either synthetic or semi-synthetic. Synthetic data are information that is artificially manufactured rather than generated by real-world events [37]. The generation of artificial data usually requires the scanning, calculation, and referencing of the data, the data model often has many dependencies between the data.

Consider one bulk load of data into the database: loading of the amount of artificial data would last pretty long since all constraints need to be checked in process of inserting the data. Usually, before bulk load, all constraints are temporarily disabled, and after the load, the constraints are enabled again. If all constraints are followed in data generation there will be no problems to enable all constraints back, but if some data are not aligned with the constraints, the whole process of data generation could be the waste of time. Having this in mind, if the bulk of data can be verified after the generation, but before the load, it will surely evade these wastes of time. Figure 11 shows the source code example of the VSL mounted on the framework engine used for the data verification. The grouping is implemented by placing the `requirement` tags inside the `group` tags, which makes a cyclic check on grouping not necessary. Uniqueness is implemented with the attribute `id` and importance is defined in two levels: minor and major with the attribute `severity`. Dependency is represented with the `depends` tag with `ref` attribute that points to the requirement. This tag can go inside both: the `group` and the `requirement` tag. For `depends` tag, we need the cyclic dependency check.

```
<group id="ScheduleTable_1" >
  <depends ref="CfgTable_1"/>
  <requirement id="3029553" severity="major" >
   <depends ref="3029872"/>
   <depends ref="3029873"/>
   <assertSizeRangeRule variable="ScheduleTable"
    minRows="0"
    maxRows="64 * lcm(CfgTable.scheduleCount)"/>
  </requirement>
</group>
```

Figure 11. Source code example on data verification

Within the requirement tag, there can be as many as possible assertion tags that check different aspects of data. There are 27 different assertions with the additional controls such as `iterate` and `case`. Figure 11 shows the usage of the extended function `lcm` that has been implemented as the python function. Function `lcm` finds the least common multiple for an array of numbers. What can't be seen in the example is the fact that any member tag of the requirement can be conditioned with an additional tag `precondition`.

V. THE RESULTS

The most important result of the verification-specific language developed using the framework is the fact that it covers all aspects of the requirement specification and verification, which is not the case with any of the related works, as Table II shows. The framework's ultimate goal is to cover all aspects of the requirement verification either by implementing it or by making it mandatory to develop, as a list of to-dos. Apart from the obvious result, every of the four presented implementation examples has its own results.

A. Previous Research Replicated with Framework Engine

The approximately 100 extra lines of code appeared as a direct consequence of the usage of the framework (for implementation of Requirement templates, and for calling other framework's functions). Table III compares the measures during the development of the conventional approach and framework approach. It needed fewer lines of code to write and therefore less time to spend to develop the same DSL. The test coverage on both approaches is the same and the ratio of the number of lines of code used for testing and the number of production lines of code differs a bit showing that there is less need for testing, which means that the framework solution is less prone to errors, since it follows the underlying architectural rules.

TABLE III. COMPARISON ON TWO SAME DSLS DEVELOPED WITH AND WITHOUT FRAMEWORK

Measure	Orthodox approach	Novel approach
Lines of Code (LoC)	4000	1500
Test. LoC / Prod. LoC	5:3	3:2
Time spent [days]	150	85

B. Enhancement of the Existing Verification Process

In this case, the focus is not on the efficiency of the developing process, but on the robustness of the solution and the execution performance. When the requirement needs to be changed (removed or added), the original verification process needs additional time for the manual requirement reordering in the verification process. The enhanced verification needs no additional time for reordering the processes. Table IV shows the measured time losses on the original verification process based on different verification sizes, due to various requirement changes. It shows constant time loss growth when the size of the verification is growing.

TABLE IV. TIME LOSSES ON DIFFERENT VERIFICATION SIZES

Type of the change on the requirement	Verification size [no. requirements]		
	50	130	290
	Average time loss [hours]		
Updating	8	20	42
Removing	7	21	43
Adding	8	17	31

Without any additional functionality for preconditions and dependencies, the original execution process runs longer, since failed verification of depending requirements does not bypass the execution of the verification. On the enhanced side, the preconditions and dependencies speed up the verification process. Table V compares the average execution time on three different sizes of the verification processes for both original and enhanced solutions. No matter what the size of the verification process is the enhanced solution shows faster execution, since whenever one requirement fails, the enhanced solution skips all depending on requirement verification. It appeared that the time savings on skipping the requirements are greater than the time losses on ordering the requirements (original solution orders requirements statically, before the execution).

TABLE V. AVERAGE EXECUTION TIMES ON DIFFERENT SIZED VERIFICATIONS

Verification size [no. requirements]	Average execution time [ms]	
	Original solution	Enhanced solution
50	14855	8140
130	31568	22986
290	78280	53981

C. Extending the Specification Language

The extension of the current requirement-specific language took approximately 80 days of work with 1500 lines of code for a skilled developer, already introduced into the framework engine. The base specification language took about 200 days of work and additional 3200 lines of code to be built.

D. Verification of Generated Data

The results show that verification of generated data makes **15% of the total time** of a process of generation and loading data. It means that the data verification for usual purposes such as loading the data for the benchmarking may not be useful. If the system that uses generated data can work without "bad" data, it makes no sense to use the data verification. But in cases when only all data makes a complete and relevant picture, the verification of the data makes sense.

VI. CONCLUSION

The Framework that has been proposed supports the implementation of VSL with all properties and functionalities mandatory for requirement verification. It has been presented that the framework followed and enforced the IEEE guides, standards, and best practices to check all aspects of the software requirements that were neither implemented nor checked by the validation process: correctness, completeness, traceability, dependency, importance, and uniqueness.

With the given results it has been proven that the framework enhances the developer's productivity, as well as software quality:

- The framework engine has made the development of the VSL more efficient. The results have shown that it needs less time to develop the same language by using the framework engine;

- The same example has reasoned better code quality. It is because the testing/product LoC is slightly smaller in the framework's approach;
- The results of the second example have shown better robustness and better performances of the framework-based solution;
- Third example has shown: By investing a fraction more of time invested in the creation of the requirement specification language, it is possible to implement the verification ability within the same language;
- Usage of the VSL improves the reliability of the data generators. It is extremely important in cases of safety-critical data generation.

REFERENCES

[1] "IEEE Standard for system and software verification and validation," in IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004), pp.1-223, 25 May 2012. doi:10.1109/IEEESTD.2012.6204026

[2] "IEEE Draft Guide: Adoption of the project management institute (PMI) standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition)," in IEEE P1490/D1, May 2011, pp. 1-505, 30 June 2011. doi:10.1109/IEEESTD.2011.5937011

[3] B. Nuseibeh, and S. Easterbrook, “Requirements engineering: A roadmap,” in Proc. of the Conf. on the Future of Software Engineering, Limerick, Ireland, 2000, pp. 35–46. doi:10.1145/336512.336523

[4] A. Morkevicius and N. Jankevicius, "An approach: SysML-based automated requirements verification," 2015 IEEE Int. Symp. on Systems Engineering (ISSE), 2015, pp. 92-97. doi:10.1109/SysEng.2015.7302739

[5] “IEEE Recommended practice for software requirements specifications,” in IEEE Std 830-1998, Reaffirmed 9-12-2009, vol., no., pp.1-40, 1998. doi:10.1109/IEEESTD.1998.88286

[6] L. Lengyel et al., "Quality assured model-driven requirements engineering and software development," in The Computer Journal, vol. 58, no. 11, pp. 3171-3186, 2015. doi:10.1093/comjnl/bxv051

[7] R. Young, The requirements engineering handbook, pp.130-150, Artech, 2003

[8] P. Rempel and P. Mäder, "A quality model for the systematic assessment of requirements traceability," 2015 IEEE 23rd Int. Requirements Engineering Conference (RE), 2015, pp. 176-185. doi:10.1109/RE.2015.7320420

[9] F. Wang et al., "An approach to generate the traceability between restricted natural language requirements and AADL models," in IEEE Transactions on Reliability, vol. 69, no. 1, pp. 154-173, 2020. doi:10.1109/TR.2019.2936072

[10] J. P. Tolvanen, S. Kelly, “How domain-specific modeling languages address variability in product line development: investigation of 23 cases,” 3rd Int. Systems and Software Product Line Conf. , 2019, pp. 155-163. doi:10.1145/3336294.3336316

[11] M. Fowler, "A pedagogical framework for domain-specific languages," in IEEE Software, vol. 26, no. 4, pp. 13-14, 2009. doi:10.1109/MS.2009.85

[12] J. P. Tolvanen, S. Kelly, Domain-specific modeling enabling full code generation, pp. 34-41, A Wiley-Interscience, 2008

[13] D. Ghosh, DSLs in Action, Greenwich, pp. 220-300, Manning Co, 2011

[14] “ITU-T, Recommendation Z.151 (10/18), User Requirements Notation (URN) – Language definition”, pp.1-140, 2018

[15] D. Amyot, L. Logrippo and M. Weiss, “Generation of test purposes from Use Case Maps,” Computer Networks, vol. 49, no. 5, pp. 643-660, 2005. doi:10.1016/j.comnet.2005.05.006

[16] D. de Almeida Ferreira and A. R. da Silva, "RSLingo: An information extraction approach toward formal requirements specifications," 2012 Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE), 2012, pp. 39-48. doi:10.1109/MoDRE.2012.6360073

[17] N. A. Ali, A. A. Mirghani and A. Y. Ibrahim, "Alneelain: A formal specification language," 2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE) , 2017, pp. 1-9. doi:10.1109/ICCCCEE.2017.7867678

[18] A. Mili and F. Tchier, Software Testing, concepts and operations, pp. 38-39, Wiley, 2015

[19] L. C. Hong and L. T. Ming, "To enable formal verification of semi-formal requirements by using pre-defined template and mapping rules to map to Promela specification to reduce rework," 2010 Int. Symposium on Information Technology, 2010, pp. 1393-1397. doi:10.1109/ITSIM.2010.5561453

[20] P. H. Feiler, B. A. Lewis and S. Vestal, "The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems," IEEE Conference on Computer Aided Control System Design, 2006, pp. 1206-1211. doi:10.1109/CACSD-CCA-ISIC.2006.4776814

[21] Z.-B.Yang et al., “From AADL to timed abstract state machines: A verified model transformation,” Journal of Systems and Software, vol. 93, pp. 42-68, 2014. doi:10.1016/j.jss.2014.02.058

[22] G. Behrmann, A. David, K. G. Larsen, “A tutorial on Uppaal,” in Formal Methods for the Design of Real-Time Systems, Berlin, Germany: Springer, vol.3185, pp. 200-236, 2004. doi:10.1007/978-3-540-30080-9_7

[23] D. Cofer, et al., “Compositional verification of architectural models,” in NASA Formal Methods, vol. 7226, pp. 126-140, 2012. doi:10.1007/978-3-642-28891-3_13

[24] G. Sutcliffe and C. Suttner, “The TPTP problem library,” Journal of Automated Reasoning, vol. 21, pp. 177-203, 1998. doi:10.1023/A:1005806324129

[25] Z. Zuo and M. Zhou, "Web Ontology Language OWL and its description logic foundation," Proc. of the Fourth Int. Conference on Parallel and Distributed Computing, Applications and Technologies, 2003, pp. 157-160. doi:10.1109/PDCAT.2003.1236278

[26] C. -. Liu, K. -. Chang, J. J. -. Chen and S. -. Hung, "Ontology-based context representation and reasoning using OWL and SWRL," 2010 8th Annual Communication Networks and Services Research Conference, 2010, pp. 215-220. doi:10.1109/CNSR.2010.22

[27] N. E. Fuch, “First-order reasoning for attempto controlled english,” Int. Workshop on Controlled Natural Language, CNL 2010: Controlled Natural Language, Springer-Verlag Berlin, 2012, pp. 73 – 94. doi:10.1007/978-3-642-31175-8_5

[28] P. Clark, W. R. Murray, P. Harrison, and J. Thompson, “Naturalness vs. predictability: A key debate in controlled languages”. Controlled Natural Language: Proc. of the Work. on Controlled Natural Language, 2009, vol. 5972, pp. 65-81. doi:10.1007/978-3-642-14418-9_5

[29] S. Thummalapenta, S. Sinha, N. Singhania and S. Chandra, "Automating test automation," 2012 34th International Conference on Software Engineering, 2012, pp. 881-891, doi: 10.1109/ICSE.2012.6227131

[30] A. Miller, B. Kumar and A. Singhal, "Photon: A domain-specific language for testing converged applications," 33rd Annual IEEE International Computer Software and Applications Conference, 2009, vol. 2, pp. 269-274. doi:10.1109/COMPSAC.2009.143

[31] M. Bernardino, A. F. Zorzo and E. M. Rodrigues, "Canopus: A domain-specific language for modeling performance testing," 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, pp. 157-167. doi:10.1109/ICST.2016.13

[32] S. Pakin, "The design and implementation of a domain-specific language for network performance testing," in IEEE Trans. on Parallel and Distributed Systems, vol. 18, no. 10, pp. 1436-1449, 2007. doi:10.1109/TPDS.2007.1065

[33] A. Dwarakanath, D. Era, A. Priyadarshi, N. Dubash and S. Podder, "Accelerating Test Automation through a Domain Specific Language," 2017 IEEE Int. Conf. on Software Testing, Verification and Validation, 2017, pp. 460-467. doi:10.1109/ICST.2017.52

[34] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, S. Schulz, A. Wiles, "Basic TTCN□3," in An Introduction to TTCN-3, Wiley, 2010, pp.25-44. doi:10.1002/9780470977903.ch3

[35] A. A. Giordano, A. H. Levesque, "Getting started with Simulink," in Modeling of Digital Communication Systems Using SIMULINK, Wiley, 2015, pp.1-26. doi:10.1002/9781119009511.ch1

[36] S. Popic, V. Komadina, R. Arsenovic and M. Stepanovic, "Implementation of the simple domain-specific language for system testing in V-Model development lifecycle," 2020 Zooming Innovation in Consumer Technologies Conference, 2020, pp. 290-294. doi:10.1109/ZINC50678.2020.9161781

[37] S. Popić, B. Pavković, I. Velikić and N. Teslić, "Data generators: a short survey of techniques and use cases with focus on testing," 2019 IEEE 9th Int. Conf. on Consumer Electronics (ICCE-Berlin), 2019, pp. 189-194. doi:10.1109/ICCE-Berlin47944.2019.8966202