PyRDV: a Python-based solution to the requirements traceability problem

Fernando Gabriel Orge
Allegro MicroSystems, Buenos Aires, Argentina
forge@allegromicro.com

Abstract - **This paper introduces the PyRDV tool, which helps IC (Integrated Circuit) developers find potential coverage holes or unimplemented requirements without running simulations. PyRDV is not just a Python-based software tool but a complete solution consisting of (1) A theoretical framework to prove design completeness, (2) A detailed workflow for IC developers based on GitLab issues, (3) A Python package to collect all the necessary information from the GitLab issues and (4) A CI/CD (Continuous Integration / Continuous Deployment) service to periodically check for sign-off metrics. Thus, this work presents an alternative solution to the requirements traceability problem and contributes to RDV (Requirement-driven Verification) methodologies, proposing detailed criteria to guarantee the implementation of all the requirements and the verification of all the specifications. The theoretical framework defines a common language for all the IC developers, preventing ambiguities in the information flow. The GitLab platform is the unique source of truth among all developers since it gathers requirements, specifications and verification plans. The GitLab CI/CD service also helps publish all the sign-off reports and metrics on the GitLab platform. As a result, developers will benefit from this prompt information because it will help them to quickly adapt to requirement changes and optimize design and verification resources.**

I. Introduction

In a typical IC (Integrated Circuit) design process, system architects, digital designers and verification engineers interact with each other to create a new product. We can think of this interaction as two consumer-producer problems, one involving system architects and designers and the other involving designers and verification engineers. In the first problem, system architects produce requirements consumed by designers to create specifications. On the other hand, verification engineers consume those specifications to create verification elements, such as test cases, assertions and functional coverage. Therefore, a successful design will meet the following conditions: designers must implement all the requirements, and verification engineers must verify all specifications. Although this might be a simple and easy-to-understand statement, it can be a big headache for IC developers during their workday. One of the main concerns for IC developers is the complexity and the size of current designs, which leads to an increase in the amount of requirements to meet. Market dynamics also generate slight variations in product requirements that developers must adapt to during the design process. These circumstances are potential pitfalls for developers since they could incur a coverage hole or even the non-implementation of a requirement. Another potential pitfall in the design process is the flow of information, which can be erratic when the number of people involved in a project gets larger. From this type of situation arises the need for an automated tool that allows us to watch the flow of information, focused on requirements traceability.

II. Related Work

Requirements-driven Verification (RDV) is a widely used methodology in the semiconductor industry to ensure successful product development. Many papers have been published over the last decade explaining the advantages and disadvantages of this methodology. The authors in [1] emphasize the relevance of RDV in automotive projects that need to comply with ISO26262 and propose a solution based on a centralized SQL database. An alternative solution is presented in [2], where requirements are managed in a proprietary platform called DOORS [3], while the verification-related data is managed using MS Excel. Then, they bridge the gap between requirements and verification through a proprietary check-and-update mechanism. The authors in [4] agree with the earlier works, emphasizing the necessity of a metrics-based methodology and requirements traceability. They also emphasize the need for a unified platform to create the verification plan and to store the coverage metrics. The authors in [5] highlight the relevance of compliance with ISO26262 and provide examples of how RDV is complementary to other verification methodologies based on metrics. Lastly, the papers [6] and [7], while not primarily focused on RDV,

illustrate how to use Python as a great complement to enhance current verification methodologies. In summary, these recent papers stress the essential role of RDV in successful project development. They also provide various approaches to the requirement traceability problem and illustrate how to complement RDV methodology with other tools and languages, such as Python.

Throughout this work, we will present an alternative solution to the RDV problem. Our approach involves using GitLab [8] as a unique source of truth for all developers, as opposed to other solutions based on SQL or DOORS. We will demonstrate that GitLab is not only effective for managing requirements information but also for managing specifications and verification plans. We will also propose using a continuous integration service as an automated mechanism for collecting progress metrics and generating reports. Finally, by introducing our Python solution called PyRDV, we will reinforce the concept of using Python-based solutions to improve and optimize workflows and design processes.

III. The PyRDV Solution

PyRDV is an acronym for Python Requirement-Driven Verification. The rationale behind this solution is simple. If verification guarantees proper feature implementation, how can we ensure we have verified everything we need to? In this section, we show a singular and a generalized solution to the traceability problem. We use this solution to guarantee that we verify all the specifications and implement all the requirements. Moreover, this generalized solution allows us to trace the verification elements back to requirements without losing generality.

A. Theoretical framework to prove design completeness: Singular Solution

As said before, the conditions for a successful design are all requirements must be implemented, and all specifications must be verified. We can extrapolate these two conditions to graph theory to formulate two assignment problems. One that relates requirements to specifications (or features), and the other that connects specifications to verification elements. It is possible to find some characteristic relations between these two elements, based on the assumption that each specification will be covered or verified by a verification element, namely:

(a) One-To-One Relation: This is the simplest and the preferred case where a specification is covered by only one single verification element. In this kind of situation, the feature is strongly covered since the relation is unambiguous.
(b) One-To-Many Relation: This is a special case where a verification element can cover more than one specification. This case is another case of strongly covered features and differs from the previous one, since it is the typical case of an integration test or top-level test.
(c) Many-To-One Relation: This is another special case where more than one verification element is needed to cover one single feature. Though not ideal, this is a common case, and the feature can be thought of as being not sufficiently atomic and it might require some sort of feature slicing. The feature, in this case, is said to be weakly covered.
(d) No-Related Specification: Any time a feature cannot be related to a verification element, the feature is said to be uncovered or not covered.
(e) No-Related Verification Element: Any time a verification element cannot be related to a feature, the verification element is said to be useless (in an over-engineering sense).

All these relations can be modeled by the following mapping function:

$$g : F \times V \to \{0, 1\} \; such \; that \; g(i, j) = \begin{cases} 1 & if \; feature_i \; is \; verified \; by \; verification \; element_j \\ 0 & if \; not \end{cases} \tag{1}$$

where g is the mapping function, F is the set of all the features and V is the set of all the verification elements.

If the total number of features $|F| = M$ must be verified with at least $|V| = N$ verification elements, then the mapping function g can be thought as a $M \times N$ matrix that will be filled with either ones or zeros as follows:

$$\begin{pmatrix} g(1,1) & g(1,2) & ... & g(1,N) \\ g(2,1) & g(2,2) & ... & g(2,N) \\ ... & ... & ... & ... \\ g(M,1) & g(M,2) & ... & g(M,N) \end{pmatrix} = \begin{pmatrix} 0 & 1 & ... & 0 \\ 1 & 0 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{pmatrix} \tag{2}$$

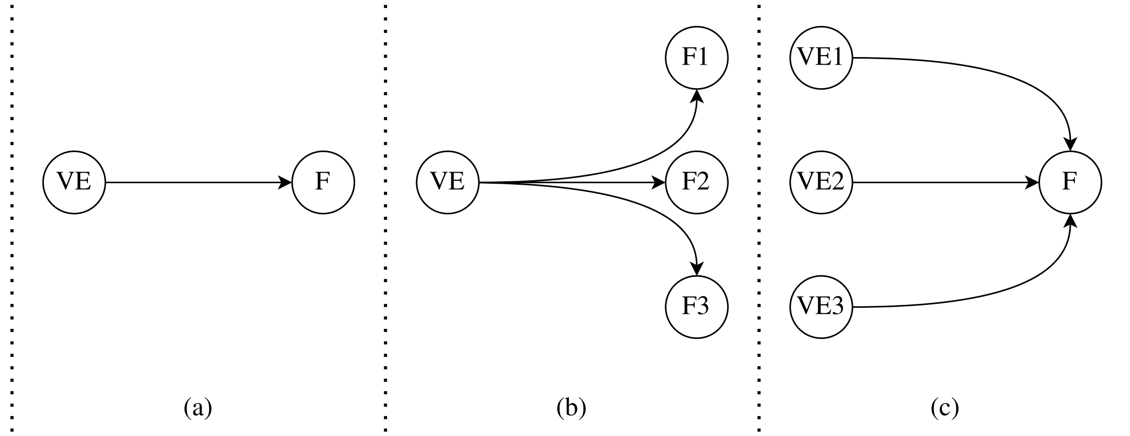


Fig. 1: Relations between Features (F) and Verification Elements (VE).
(a) One-To-One Relation. (b) One-To-Many Relation. (c) Many-To-One Relation.

Considering that each row in this matrix is related to each feature in F, if the sum of all the elements across the row is equal to 0 (zero), then the feature is "not covered" by any verification element in V. Moreover, every time this sum is equal to 1 (one), that means that the feature is covered by exactly one verification element. Since a One-To-One mapping is expected, this will be the ideal case, and the featured is said to be "strongly covered." Finally, if the sum of all the elements across the row is greater than 1 (one), that means that one single feature is covered by more than one verification element. Though not ideal, this is a common case, and the feature can be thought of as being not sufficiently atomic. The feature, in this case, is "weakly covered." All these relationships are formulated in (3), (4) and (5) respectively.

$$feature_i \ is \ considered \ as \ not \ covered \ iff \ \sum_{j \in V} g(i,j) = 0 \tag{3}$$

$$feature_i \ is \ considered \ as \ strongly \ covered \ iff \ \sum_{j \in V} g(i,j) = 1 \tag{4}$$

$$feature_i \ is \ considered \ as \ weakly \ covered \ iff \ \sum_{j \in V} g(i,j) > 1 \tag{5}$$

On the other hand, considering that each column in the above matrix is related to each verification element in V, if the sum of all the elements across the column is equal to 0 (zero), that means that the element is not related to any feature. The element is said to be "useless." Furthermore, every time this sum is equal to 1 (one), that means that the element is covering exactly one feature. Since a One-To-One mapping is expected, this will be the ideal case, and the element is "strongly linked." Finally, if the sum of all the elements across the column is greater than 1 (one), that means that one single element is covering more than one feature. Though not the ideal case, this is a common case, and the features can be thought of as being simply enough to be grouped together and being covered by a single element. The element, in this case, is said to be "linked." All these relationships are formulated in (6), (7) and (8) respectively.

$$verification \ element_j \ is \ considered \ to \ be \ useless \ iff \ \sum_{i \in F} g(i,j) = 0 \tag{6}$$

$$verification \ element_j \ is \ considered \ to \ be \ strongly \ linked \ iff \ \sum_{i \in F} g(i,j) = 1 \tag{7}$$

$$verification \ element_j \ is \ considered \ to \ be \ linked \ iff \ \sum_{i \in F} g(i,j) > 1 \tag{8}$$

To conclude, the product is fully covered when each feature is (strongly or weakly) covered and there are no useless verification elements. We refer to this as the completeness condition and it is formally expressed by (9).

$$\sum_{j \in V} g(i,j) \geq 1, \ \forall i \in F \quad and \quad \sum_{i \in F} g(i,j) \geq 1, \ \forall j \in V \tag{9}$$

B. Theoretical framework to prove design completeness: General Solution

Let T be a set of elements that needs to be covered by elements in another set called B (T stands for Top set and B for Bottom set). Then, the mapping function shown in (1) is redefined as

$$g : T \times B \to \{0, 1\} \; such \; that \; g(i, j) = \begin{cases} 1 & if \; t_i \; is \; covered \; by \; b_j \\ 0 & if \; not \end{cases} \tag{10}$$

Where t_i is any element in T and b_j is any element in B. The relationships previously seen from (3) to (8) are redefined as follows:

$$t_i \; is \; considered \; as \; not \; covered \; iff \; \sum_{j \in B} g(i, j) = 0 \tag{11}$$

$$t_i \; is \; considered \; as \; strongly \; covered \; iff \; \sum_{j \in B} g(i, j) = 1 \tag{12}$$

$$t_i \; is \; considered \; as \; weakly \; covered \; iff \; \sum_{j \in B} g(i, j) > 1 \tag{13}$$

$$b_j \; is \; considered \; to \; be \; useless \; iff \; \sum_{i \in T} g(i, j) = 0 \tag{14}$$

$$b_j \; is \; considered \; to \; be \; strongly \; linked \; iff \; \sum_{i \in T} g(i, j) = 1 \tag{15}$$

$$b_j \; is \; considered \; to \; be \; linked \; iff \; \sum_{i \in T} g(i, j) > 1 \tag{16}$$

The completeness condition shown in (9) is redefined as follows:

$$\sum_{j \in B} g(i, j) \geq 1, \; \forall i \in T \quad and \quad \sum_{i \in T} g(i, j) \geq 1, \; \forall j \in B \tag{17}$$

Now, it is easy to see that the solution shown in the previous section is one that considers the set T as the set of all the specifications that needs to be covered by the set B, that is, the set of all the verification elements. Finally, if T is now the set of all the requirements that needs to be implemented by the specification (or features) in B, we then have the solution to the requirement traceability problem. Moreover, it is possible to track the requirements against the verification elements (instead of features), if we replace the set of features by the set of verification elements.

Since we want to have all features verified and all requirements implemented, we will have two completeness conditions for our projects. Using (17), we will solve the features-to-verification-elements completeness and the requirements-to-features completeness, respectively.

C. Workflow for IC developers based on GitLab

The proposed workflow uses GitLab [8] as a unique source of truth. Using it allows everyone involved in the project to see and consume the information they need from a centralized location. Developers use GitLab issues to publish any information. They typically proceed in the following way: system architects create issues describing the product requirements and add a convenient label to tag them (i.e., REQUIREMENT). Designers create issues detailing the specifications and add a convenient label to tag them (i.e., FEATURE). In addition, as part of the description, they write down which requirement they are implementing (e.g., something as simple as a comment that says ”implements requirement X”). Verification engineers create issues describing the verification plan elements and add a suitable label to tag them (i.e., VPLAN ELEMENT). They also write down which specification they are covering (e.g., something as simple as a comment that says ”verifies feature Y”). The PyRDV Tool will treat these standard comments as regular expressions when analyzing issues content.

This whole process of creating and publishing issues is depicted in the top part of Fig. 2. The workflow has two main advantages: since GitLab is a central location for all the information, it supplies a straightforward way for everyone to find the information they need. The other significant advantage are the labels, which makes filtering and searching tasks much easier for the developers and the PyRDV Tool.

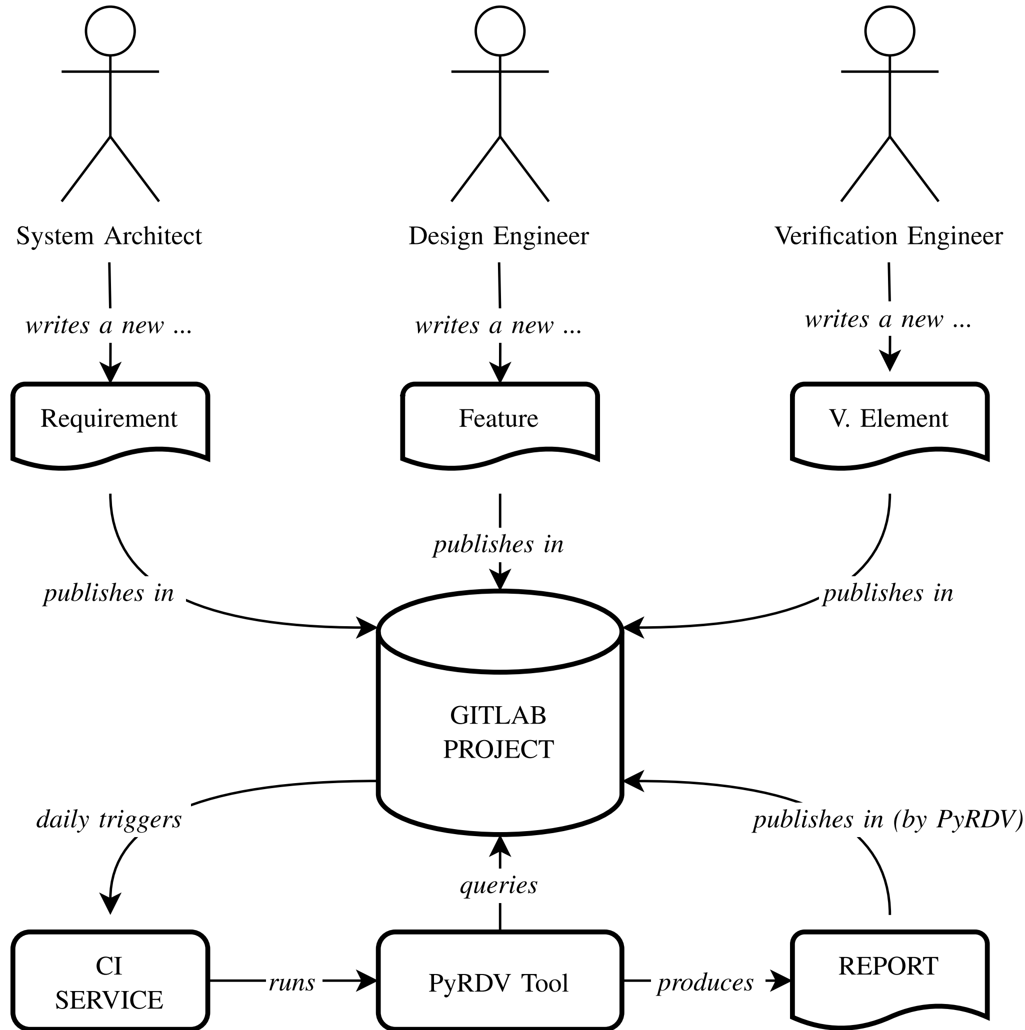


Fig. 2: Complete workflow.

D. GitLab CI/CD Service

GitLab CI/CD [9] is a continuous integration and continuous deployment tool typically used by software developers to automate the process of building, testing, and deploying software. However, the software industry is not the only one that can benefit from this tool. The semiconductor industry can benefit from GitLab CI/CD as well. This tool integrates naturally with GitLab, and developers can use it to trigger any number of automated tasks regularly or any time a developer pushes a new commit into the repositories. These automated tasks are known as pipelines and may include compilation, elaboration or simulation tasks.

By using GitLab CI/CD service, we can trigger the PyRDV Tool regularly (daily or weekly, depending on project needs). This automated way to check for completeness significantly reduces the developers' efforts as they no longer need to verify if they met all the requirements consistently. These periodic checks will generate reports to show project evolution and mapping inconsistencies. Having this prompt information allows us to save time and resources since we can quickly adapt our schedules to cover those unimplemented requirements or unverified specifications. We show a sample report in the following section.

E. The PyRDV Tool

The PyRDV tool is an in-house solution developed in Python language. This solution is based on a well-known design pattern called Chain of Responsibility [10], which allows us to organize the source code in a convenient way. One of the key benefits of this pattern lies in its flexibility, enabling the introduction of new tasks or the

removal of existing ones without causing any disruption to the client code. The Chain of Responsibility operates on a basic principle: each worker within the chain will do one single and unique task. Upon completion of their respective task, each worker delivers a result that might be used by any of the subsequent workers in the chain. The Chain of Responsibility is depicted in Fig. 3, which illustrates the sequential flow of tasks.

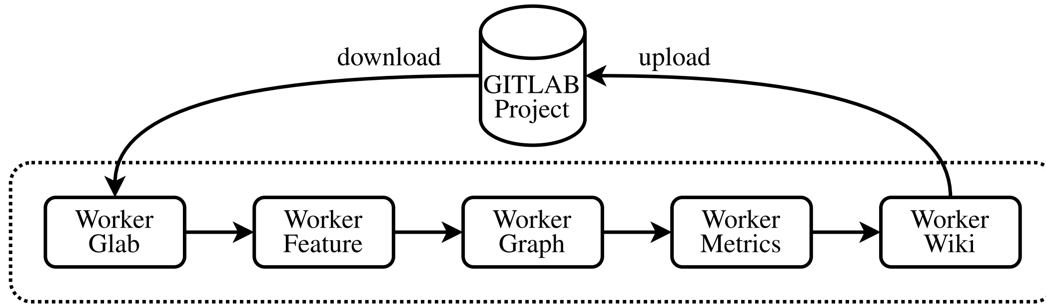


Fig. 3: PyRDV Tool - Chain of Responsibility.

We developed the workers according to the class hierarchy shown in Fig. 4. The Worker class declares a common interface to all workers, which defines the methods to execute the tasks and to queue workers in the chain. Then, the WorkerAbstract class acts as a base class for the rest of the workers. We used this intermediate class to make additional methods accessible to all workers and to keep the interface intact when updating the source code. Finally, the rest of the workers are extensions to the base class, and they override the methods defined at the interface. At the top of the hierarchy is the ABC class, which allows the definition of abstract classes in Python. The ABC class is defined in the ABC Python package [11].

The Worker "Glab" is responsible for executing the first task in the chain, which consists of connecting to the GitLab servers and retrieving information related to all the issues tagged by the developers. We have used the python-gitlab package [12] to facilitate the development of this worker.

The Worker "Feature" is responsible for analyzing the content of the issues collected before. The name of this worker resembles the idea of "feature extraction." Therefore, this worker will look at the content of each issue for the regular expressions stated in the workflow. It will establish the relationships between the elements (i.e., requirements, features, and verification elements) by extracting those regular expressions.

The Worker "Graph" uses known Python packages such as networkx [13] and matplotlib [14] to generate the graph plots that we will publish as part of the results report. Similarly, the Worker "Metrics" is responsible for computing all the metrics we will post in the report. These metrics include trend plots to show project evolution and classifications of each extracted issue. We obtain these classifications by computing all the equations explained during the theoretical framework.

Finally, the Worker "Wiki" is responsible for organizing the information generated by previous workers and drafting a report in Markdown [15]. The benefit of using Markdown is that it is the preferred language for GitLab pages, and therefore, its visualization is more comfortable for all users.

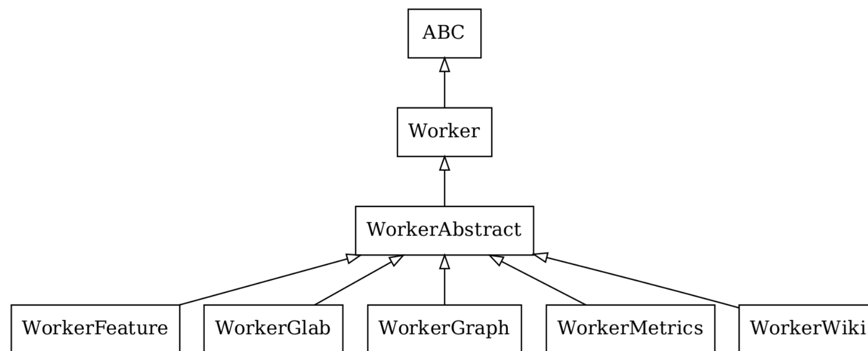


Fig. 4: PyRDV Tool - Workers Class Hierarchy.

IV. Sample Case

We present a sample case that exemplifies the proposed solution. The case contains eight requirements, eight features or specifications and eight verification elements. It is worth recalling that the goal is to have all features verified and all requirements implemented. Therefore, the goal is to guarantee the completeness condition shown in (17).

Fig. 5a, Fig. 5b and Fig. 5c are visual representations of the mapping function explained by (10). We see in Fig. 5a which features (blue dots) implement the requirements (red dots). It is easy to notice that requirement #24 is unimplemented while feature #1 is unlinked. Similarly, the graph in Fig. 5b shows how we linked verification elements to the specifications. In this graph, PyRDV could not link feature #7 and verification element #15 to any other element. Finally, we show in Fig. 5c the complete trace of the verification elements (green dots) back to the requirements (red dots). If we consider the case of requirement #10, we can see that it is implemented by feature #3, which is verified by element #11. Therefore, Fig. 5c shows a link between the requirement #10 and the verification element #11. The visual representation is notably helpful and is very descriptive. However, when the number of elements in each set is large, graph visualization can become cumbersome. For this reason, the report also includes a detailed version of these relationships.

Fig. 6 shows a typical report published by the PyRDV Tool. Fig. 6a shows the progress of requirements implementation, while Fig. 6b shows the progress of verification tasks. They also show whether we reach 100% completeness or not. The table in Fig. 6c shows the links between requirements and features and their respective status. The status column allows us to quickly recognize potential pitfalls, such as the absence of a link (e.g., requirement #24 or feature #1), or cases of multiple associations, which we have defined as weak links (e.g., requirement #22 and #23 and feature #2). Similarly, the table in Fig. 6d shows the relationship between the features and the verification elements.

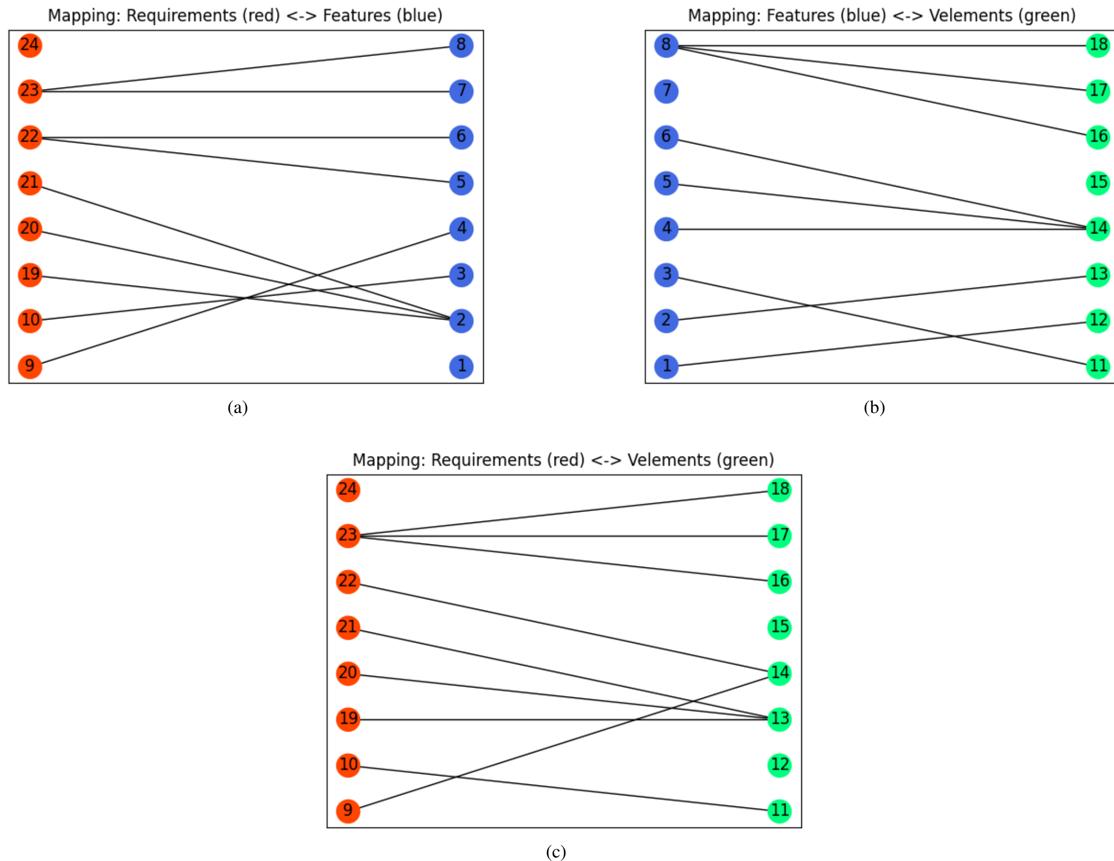
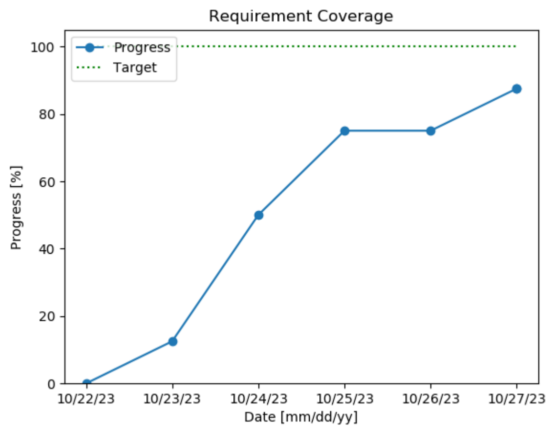
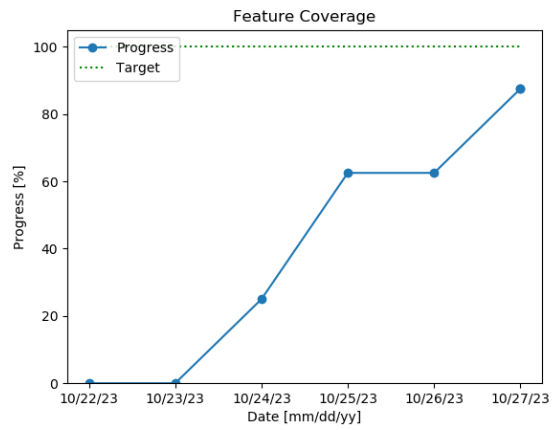


Fig. 5: Graphical representation of each mapping problem.
(a) Requirement-To-Feature Mapping. (b) Feature-To-Verification Mapping. (c) Requirement-To-Verification Mapping.



(a)



(b)

Requirements

Requirement	Related Features	GRADE	STATUS
#9	#4	STRONGLY COVERED	✓
#10	#3	STRONGLY COVERED	✓
#19	#2	STRONGLY COVERED	✓
#20	#2	STRONGLY COVERED	✓
#21	#2	STRONGLY COVERED	✓
#22	#5 #6	WEAKLY COVERED	!?
#23	#7 #8	WEAKLY COVERED	!?
#24		NOT COVERED	✗

Features

Feature	Related Requirements	GRADE	STATUS
#1		NOT LINKED	✗
#2	#21 #20 #19	WEAKLY LINKED	!?
#3	#10	STRONGLY LINKED	✓
#4	#9	STRONGLY LINKED	✓
#5	#22	STRONGLY LINKED	✓
#6	#22	STRONGLY LINKED	✓
#7	#23	STRONGLY LINKED	✓
#8	#23	STRONGLY LINKED	✓

(c)

Features

Feature	Related Vplan Elements	GRADE	STATUS
#1	#12	STRONGLY COVERED	✓
#2	#13	STRONGLY COVERED	✓
#3	#11	STRONGLY COVERED	✓
#4	#14	STRONGLY COVERED	✓
#5	#14	STRONGLY COVERED	✓
#6	#14	STRONGLY COVERED	✓
#7		NOT COVERED	✗
#8	#16 #17 #18	WEAKLY COVERED	!?

Vplan Elements

Vplan Element	Related Features	GRADE	STATUS
#11	#3	STRONGLY LINKED	✓
#12	#1	STRONGLY LINKED	✓
#13	#2	STRONGLY LINKED	✓
#14	#4 #5 #6	WEAKLY LINKED	!?
#15		NOT LINKED	✗
#16	#8	STRONGLY LINKED	✓
#17	#8	STRONGLY LINKED	✓
#18	#8	STRONGLY LINKED	✓

(d)

Fig. 6: PyRDV Sample Report. (a) (c) Requirement-To-Feature Mapping: coverage trend and detailed relations. (b) (d) Feature-To-Verification Mapping: coverage trend and detailed relations.

V. CONCLUSIONS

We have presented a novel framework based on GitLab, a common language for RDV methodology and a generalized solution to the requirement traceability problem. Our approach uses GitLab Projects as the sole source of truth for all design-related information, including requirements, specifications, and verification elements. We also developed the PyRDV for gathering and processing information from GitLab and generating reports.

To conclude, we can affirm that the main contribution of this work to the existing RDV methodologies is to prove GitLab as a way to centralize and manage requirements information, specifications and verification plans. Furthermore, this work leverages Python-based software solutions and CI/CD services to improve the design and verification workflow. Another relevant aspect of this work is that it takes advantage of the fact that GitLab is already a tool in use by the company, eliminating the need for developers to adapt to a new third-party tool. Nonetheless, we want to emphasize that our framework is not limited to GitLab and can be applied to any platform that offers both an issue-tracking capability and a continuous integration service. Finally, we want to stress that we chose Python for our software solution due to its simplicity and flexibility.

We believe this approach can significantly improve the effectiveness and efficiency of verification planning and the overall design process, given that, with this solution, we can ensure that no matter the number of requirements and features to implement, we will always be able to guarantee that all of them are adequately covered. It also allows us to find potential coverage holes faster, thanks to having timely knowledge of the verification progress. Nevertheless, there is still room for improvement. We are planning to use artificial intelligence techniques to predict the execution time of future projects based on the information collected by the PyRDV Tool.

REFERENCES

[1] S. J. Chapman, D. Galpin, and M. Bartley, "Requirements driven verification methodology (for standards compliance)," *Design and Verification Conference and Exhibition Europe (DVCon)*, 2014.

[2] W. Tibboel and J. Vink, "Closing the loop from requirements management to verification execution for automotive applications," *Design and Verification Conference and Exhibition Europe (DVCon)*, 2015.

[3] IBM, "Ibm engineering requirements management," https://www.ibm.com/products/requirements-management, (accessed Dec. 22, 2023).

[4] M. Rohleder, C. Rottgermann, S. Ruttiger, J. Brennan, M. Graham, and R. Oddone, "Enhancements of metric driven verification for the iso26262," *Design and Verification Conference and Exhibition Europe (DVCon)*, 2016.

[5] W. Tibboel, H. v. Orsouw, and S. Han, "An efficient requirements-driven and scenario-driven verification flow," *Design and Verification Conference and Exhibition Europe (DVCon)*, 2017.

[6] M. Cieplucha and W. Pleskacz, "New constrained random and metric-driven verification methodology using python," *Design and Verification Conference and Exhibition USA (DVCon)*, 2017.

[7] S. Hristozkov, J. Pallister, and R. Porter, "No country for old men – a modern take on metrics driven verification," *Design and Verification Conference and Exhibition USA (DVCon)*, 2021.

[8] GitLab, "Gitlab is the most comprehensive ai-powered devsecops platform," https://about.gitlab.com/why-gitlab/, (accessed Dec. 22, 2023).

[9] GitLab, "Get started with gitlab ci/cd," https://docs.gitlab.com/ee/ci/, (accessed Dec. 22, 2023).

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[11] Python, "abc — abstract base classes," https://docs.python.org/3/library/abc.html, (accessed Dec. 22, 2023).

[12] python-gitlab team©, "python-gitlab," https://python-gitlab.readthedocs.io/en/stable/, (accessed Dec. 22, 2023).

[13] NetworkX-developers©, "Networkx network analysis in python," https://networkx.org/, (accessed Dec. 22, 2023).

[14] M. development team©, "Matplotlib: Visualization with python," https://matplotlib.org/, (accessed Dec. 22, 2023).

[15] GitLab, "Gitlab flavored markdown (glfm)," https://docs.gitlab.com/ee/user/markdown.html, (accessed Dec. 22, 2023).