Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
AG Softwaretechnik

Graph-Based Analysis of Redundancies and Conflicts between User Stories

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science

vorgelegt von
Amir Rabieyan Nejad
Matrikel-Nr: 3350269

28. Juli 2024

Selbstständigkeitserklärung

Hiermit versichere ich, Amir Rabieyan Nejad, dass ich die vorliegende Arbeit mit dem Titel *Graph-Based Analysis of Redundancies and Conflicts between User Stories* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Masterarbeit wurde in der jetzigen oder ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen anderen Prüfungszwecken gedient.

Marburg, den 28. Juli 2024

Abstract

User stories (USs), the basic building blocks of software development, serve as precise and testable descriptions of a software's functionality. Within the dynamic framework of *agile development*, these USs become a frequently used requirements notation in agile projects[51], which is usually written informally in plain text and managed in the *product backlog*, which serves as a repository for prioritising and tracking development tasks.

As the amount of USs increases, conflicts or redundancies between them are inevitable. If a user story (US) requires the deletion of a component that is essential for the successful execution of another US, we are dealing with a conflict, or if a US (or some elements or parts of it) is a syntactic duplication of another US, we are dealing with redundancy.

In addition, changing an existing requirement or adding a new requirement to the existing product backlog in agile software development can also cause conflicts or redundancies due to changes in the needs and concerns of *system stakeholders.* This can result in a wide range of inconsistencies, as requirements are raised by multiple stakeholders involved in product development to achieve different functions.

Effectively recognising these conflicts and redundancies is fundamental for development teams. By addressing these issues, teams can provide additional value to users, adapt to changing requirements and maintain consistency between USs.

Normally, agile methods such as *Scrum* encourage cross-functional collaboration and daily stand-up meetings as mechanisms to address and mitigate redundancies and conflicts in a timely manner. However, this approach can be time and resource consuming. In cases where the backlog is very extensive, recognising existing redundancies and conflicts between USs can become a complex undertaking.

Since there is no method for automatically identifying redundancies and conflicts between requirements written in natural language in agile software development, in this thesis, we want to present two approaches for analysing redundancies and conflicts between USs.

The first approach analyses redundancies by taking annotated USs as input (instead of US text) and using *graph transformation* (GT), in particular *Henshin* and its *Conflict and Dependency Analysis* (CDA) tool, to detect potential redundancies in a syntactic way.

The second approach deals with analysing conflicts between USs, also using annotated USs as input. By utilizing *Natural Language Processing* (NLP) technique, particularly VerbNet, and a specially implemented tool, we recognize potential conflicts in a semantic way.

We apply both approaches to 19 annotated backlog datasets and create comprehensive reports for each dataset. Upon evaluation, we find that the results of both approaches are satisfactory. However, we also notice that the quality of the USs and their annotations significantly influences the effectiveness of the results.

Zusammenfassung

User Stories (USs), die Grundbausteine der Softwareentwicklung, dienen als präzise und testbare Beschreibungen der Funktionalität einer Software. Im dynamischen Rahmen der *agilen Entwicklung* werden diese USs zu einer häufig verwendeten Anforderungsnotation in agilen Projekten[51], die in der Regel informell in Klartext geschrieben und im *product backlog* verwaltet wird, das als Repository für die Priorisierung und Verfolgung von Entwicklungsaufgaben dient.

Wenn die Anzahl der USs zunimmt, sind Konflikte oder Redundanzen zwischen ihnen unvermeidlich. Wenn eine User Story (US) die Streichung einer Komponente erfordert, die für die erfolgreiche Ausführung einer anderen US unerlässlich ist, haben wir es mit einem Konflikt zu tun, oder wenn eine US (oder einige Elemente oder Teile davon) eine syntaktische Duplikation einer anderen US ist, haben wir es mit Redundanz zu tun.

Darüber hinaus kann die Änderung einer bestehenden Anforderung oder das Hinzufügen einer neuen Anforderung zum bestehenden Product Backlog in der agilen Softwareentwicklung aufgrund von Änderungen der Bedürfnisse und Anliegen der Systembeteiligten ebenfalls zu Konflikten oder Redundanzen führen. Dies kann zu einer Vielzahl von Inkonsistenzen führen, da Anforderungen von mehreren an der Produktentwicklung beteiligten Interessengruppen gestellt werden, um unterschiedliche Funktionen zu erreichen.

Die effektive Erkennung dieser Konflikte und Redundanzen ist für Entwicklungsteams von grundlegender Bedeutung. Indem sie sich mit diesen Problemen auseinandersetzen, können die Teams den Benutzern einen zusätzlichen Nutzen bereitstellen, sich an veränderte Anforderungen anpassen und die Konsistenz zwischen den USs aufrechterhalten.

Normalerweise fördern agile Methoden wie *Scrum* die funktionsübergreifende Zusammenarbeit und tägliche Stand-up-Meetings als Mechanismen, um Redundanzen und Konflikte zeitnah anzugehen und zu entschärfen. Dieser Ansatz kann jedoch zeit- und ressourcenaufwendig sein. In Fällen, in denen der Rückstand sehr groß ist, kann das Erkennen von Redundanzen und Konflikten zwischen USs zu einem komplexen Unterfangen werden.

Da es in der agilen Softwareentwicklung keine Methode zur automatischen Erkennung von Redundanzen und Konflikten zwischen in natürlicher Sprache verfassten Anforderungen gibt, möchten wir in dieser Arbeit zwei Ansätze zur Analyse von Redundanzen und Konflikten zwischen USs vorstellen.

Der erste Ansatz analysiert Redundanzen, indem er annotierte USs als Input nimmt (anstelle von US-Text) und *graph transformation* (GT), insbesondere *Henshin* und sein *Conflict and Dependency Analysis* (CDA) Tool, verwendet, um potenzielle Redundanzen auf syntaktische Weise zu erkennen.

Der zweite Ansatz befasst sich mit der Analyse von Konflikten zwischen USs und verwendet ebenfalls annotierte USs als Input. Durch die Verwendung von *Natural Language Processing* (NLP)-Techniken, insbesondere VerbNet, und einem speziell implementierten Tool, erkennen wir potenzielle Konflikte auf semantische Weise.

Wir wenden beide Ansätze auf 19 annotierte Backlog-Datensätze an und erstellen

umfassende Berichte für jeden Datensatz. Bei der Auswertung stellen wir fest, dass die Ergebnisse beider Ansätze zufriedenstellend sind. Wir stellen jedoch auch fest, dass die Qualität der USs und ihrer Annotationen die Effektivität der Ergebnisse erheblich beeinflusst.

Contents

1 Introduction

A *user story* (US) is a brief, semi-structured sentence and informal description of some aspect of a software system that illustrates requirements from the user's perspective [43]. The brief motivation statement followed the pattern: As a <*role*> I want to <*action*> , so that <*value*>.

Product backlog in agile software development acts as a repository for user stories (USs), reflecting the evolving needs and concerns of *system stakeholders*[47]. Additionally, the involvement of a *product owner* (PO) in agile development has broken down traditional barriers between development teams and end-users, fostering an environment where the product backlog becomes a central artefact guiding the development process[47].

However, there is no formal language for expressing stories or modelling backlogs from a practical point of view. Managing redundancies and conflicts between USs is increasingly important, and although agile methodologies such as *scrum* advocate collaboration and daily stand-up meetings to resolve conflicts and redundancies, the process can be resource intensive, especially with extensive backlogs.

Although USs are widely used, there are only a few methods available for evaluating and enhancing their quality.

The IEEE Recommended Practice for Software Requirements Specifications outlines eight key quality attributes for requirements[14]: correct, unambiguous, complete, consistent, ranked for importance/stability, verifiable, modifiable, and traceable. Current methods for evaluating the quality of USs include the heuristics of the INVEST framework[1] (an acronym for independent, negotiable, valuable, estimable, small, and testable) or SMART (an acronym for specific, measurable, assignable, realistic, and time-bound). Lucassen et al. offer a more detailed approach with a Quality User Story (QUS) framework based on a conceptual model for annotating USs[36], making key information explicit and defining quality dimensions semi-formally. However, this method does not adequately check for redundancies and conflicts.

In between automated support for extracting domain models from requirements artefacts such as USs play a central role in effectively supporting the detection of redundancies and conflicts between USs. Domain models are a simple way to understand the relationship between artefacts and the whole system. For example, Mosser et al. propose a model engineering method (and the associated tooling) to exploit a graph-based meta-modelling and compositional approach[40].

An important gap in current methodologies is the lack of comprehensive redundancy and conflict analysis using graph-based annotations of USs with the main motivation of shortening the feedback loop between developers and POs while supporting the iterative and incremental nature of agile development. While some tools and techniques exist for managing and prioritizing USs[46][2], they do not adequately address the problem of redundancies between USs in syntactic way. Redundancies can lead to wasted effort, inconsistencies in implementation, and ultimately, a less coherent final product. Effective

[1] http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

analysing of redundancies would ensure that similar or overlapping USs are identified and merged or eliminated early in the development process.

Furthermore, there is no framework for systematically analysing conflicts in a semantic way. Instead, the above-mentioned methods such as INVEST or SMART ensure that the USs are well defined. Conflicts between USs can arise for various reasons, e.g. due to overlapping functionalities or conflicting requirements, or ambiguity during defining USs. If left unresolved, these conflicts can significantly hinder the progress of a project.

Natural language processing (NLP) techniques offer potential advantages to improve the quality of USs and can be used to parse, extract, or analyse US's conflicts. It has been widely used to help in the software engineering domain *e.g.*, managing software requirements [5], extraction of actors and actions in requirement document [3].

Furthermore, the incorporation of computational lexicon resources like *VerbNet*[2] aids in semantic analysis, capturing linguistic and semantic data for a comprehensive understanding.

The overall target of this thesis is to introduce two well-structured workflows, one of which accelerates the automatic detection of potential redundancies using *graph transformation* (GT), *Henshin* and *conflict and dependency analysis* (CDA) tool. Another framework is used to accelerate the automatic detection of potential conflicts between USs expressed in NLP specially VerbNet and an implemented tool.

For this reason, we use the backlogs annotated with Doccano tool[3] presented by Mosser et al.[4] as the primary input [6] and apply the conflicts and redundancies analysis to them.

In order to systematically identify redundancies between USs syntactically, we use the extension CDA from Henshin in addition to model-driven transformation rules. These tools are applied to pairs of USs to determine whether they are *partially* or *fully* redundant based on predefined criteria. This process generates a detailed report highlighting potentially redundant pairs, which is then reviewed by the project team. The team assesses the results and takes the necessary actions accordingly.

For the detection of conflicts between USs in semantic way, we use VerbNet and a specially implemented tool. This approach allows us to recognise conflicts in a comprehensive manner. The generated report lists potential conflict pairs and provides valuable insights to the project team. After receiving the report, the team can review the conflicts and decide on appropriate measures to resolve them.

This paper is organised as follows: The introduction to the US and US quality assurance is given in Section 2. In Section 3 we deal with the *extracting domain models from textual requirements*, NLP and *graph transformation tool*. In Section 4, we comprehensively present our framework related to redundancy analysis. In Section 5 we present the second framework namely conflict analysis and conclude with Section 6.

[2]https://verbs.colorado.edu/verbnet
[3]https://doccano.github.io/doccano
[4]https://github.com/ace-design/nlp-stories

2 Introduction to US and US Quality Assurance

In Section 2.1 we introduce the role of USs and backlogs in agile development and the most frequently used pattern of USs is presented.

In Section 2.2, the focus is on the *quality user story* (QUS) framework and its tool AQUSA to answer the question of the existence of criteria and an automatic way to manage and identify conflicts and redundancies between USs. Following this investigation, we draw a conclusion as to whether the existing criteria are able to optimise the USs in terms of their conflicts and redundancies.

2.1 Role of User Stories and Backlogs in Agile Development

The agile software development paradigm broke the wall that classically existed between the development team and end-users. Thanks to the involvement of a *product owner* (PO) who acts as a proxy to end-users for the team, the product backlog [47] became a first-class citizen during the product development.

Furthermore, thanks to a set of USs expressing features to be implemented in the product in order to deliver value to end-users, the development teams were empowered to think in terms of added value when planning their subsequent developments. The product is then developed iteration by iteration.

Sedano et al. posited that a “product backlog is an informal model of the work to be done” [47]. A backlog implements a shared mental model among the practitioners working on a given product, acting as a boundary artefact between stakeholders. This model is voluntarily kept informal to support rapid prototyping and brainstorming sessions. Classically, backlogs are stored in project management systems, such as Jira[5] . These tools store user stories as tickets, where stakeholders write text as natural language. Meta-data (*e.g.*, architecture components, severity, quality attribute) can also be attached to the stories. However, there is no formal language to express stories or model backlogs from a state of practice point of view.

A *user story* (US) is a brief, semi-structured sentence and informal description of some aspect of a software system that illustrates requirements from the user’s perspective [43]. Large, vague stories are called epics. While USs vary widely between organizations, most observed stories included a motivation and acceptance criteria. The brief motivation statement followed the pattern: As a <*role*> I want to <*action*> so that <*value*>. This is sometimes called the *Connextra* template. The acceptance criteria followed the pattern: Given <*context*>, when <*condition*> then <*action*>. This is referred to as *Gherkin syntax*[54]. It consists of three aspects, namely aspects of *who*, *what* and *why*. The aspect of “who” refers to the system user or actor, “what” refers to the actor’s desire, and “why” refers to the reason (optional in the user story) [43].

The US components consist of the following elements[52]:

- *role*: abstract behaviour of actors in the system context; the aspect of who represents.

[5] https://www.atlassian.com/en/software/jira

- *goal*: a state or circumstance that is desired by stakeholders or actors
- *task*: specific things that need to be done to achieve goals.
- *Capability*: the ability of actors to achieve goals based on certain conditions and events.

2.2 QUS Framework and AQUSA as a Tool

Lucassen et al. [35] present a quality user story (QUS) framework consisting of 13 quality criteria that US authors should strive for. The criteria analysed determine the intrinsic quality of USs in terms of *syntax*, *pragmatics* and *semantics*. Figure 1 illustrates the structure of the agile requirements verification framework. Table 2 also shows the QUS framework, which defines 13 criteria for the quality of USs.

Based on QUS, Lucassen et al. present the automatic quality user story artisan (AQUSA) software tool for assessing and enhancing US quality automatically. Relying on NLP techniques, AQUSA detects quality defects and suggests possible remedies.

A US should follow a pre-defined, agreed template, chosen from the many templates available. In the conceptual model the skeleton of the template is called *format*, which the *role*, *means*, and optional *end(s)* are interspersed to form a US [53].

Because USs are a controlled language, the QUS framework's criteria are organize in *Lindland's* categories [32]:

- *Syntactic quality*, about the textual structure of a US without taking its meaning into account;
- *Semantic quality*, about the relationships and the meaning of (parts of) the US text;
- *Pragmatic quality*, takes into account not only syntax and semantics, but also the subjective interpretation of the US text by the audience.

By Lucassen et al. introduced quality criteria divided into two categories, namely *Individual*, which applies to single US, and *Set*, which applies to a bundle of USs. Individual criteria can be evaluated against an individual US:

well-formed

To be considered US, the core text of the request must contain a role and the expected functionality: the *means*. Looking at the US "I want to see an error message if I can't see recommendations after uploading an article". It is likely that the US author has forgotten to specify the role. The error can be fixed by adding the role: "As a member, I would like to see an error message if I cannot see recommendations after uploading an article".

Figure 1: Agile Requirements Verification Framework [35]

Atomic

A US should only concern one characteristic. Although it is common in practice, combining multiple USs into a larger, general US affects the accuracy of effort estimation[33]. For example, the US "As a user, I can click on a specific location on the map and thereby perform a search for landmarks associated with that combination of latitude and longitude" consists of two separate requests: Clicking on a location and viewing the associated landmarks. This US should be split into two parts:

- "As a user, I am able to click on a specific location on the map";
- "As a user, I am able to see landmarks associated with the combination of latitude and longitude of a specific location".

Minimal

User stories should contain a role, a means and (ideally) some goals. Any additional information such as comments, descriptions of expected behaviour or notes on testing should be noted in additional notes. View the US "As a supervisor, I would like to see the registered hours for this week (split into products and activities). See: Mock-up by Alice NOTE-first create the overview screen-then add validations". In addition to a role and resources, it includes a reference to an undefined mock-up and an indication of how the implementation should be approached. The requirements engineer should move both to separate US attributes such as the description or comments and keep only the basic story text: "As a care professional, I would like to see this week's registered hours".

Conceptually sound

The middle and end parts of a US play a special role. The middle part should capture a specific feature, while the end part expresses the rationale for that feature. Let's look at the US "As a user, I want to open the interactive map so that I can see the location of the points of interest": The purpose is actually a dependency on another (hidden) feature that is required for the purpose to be fulfilled, which requires the presence of

Criteria	Description	Individual/Set
Syntactic		
Well-formed	A user story includes at least a role and a means	Individual
Atomic	A user story expresses a requirement for exactly one feature	Individual
Minimal	A user story contains nothing more than role, means, and ends	Individual
Semantic		
Conceptually sound	The means expresses a feature and the ends expresses a rationale	Individual
Problem-oriented	A user story only specifies the problem, not the solution to it	Individual
Unambiguous	A user story avoids terms or abstractions that lead to multiple interpretations	Individual
Conflict-free	A user story should not be inconsistent with any other user story	Set
Pragmatic		
Full sentence	A user story is a well-formed full sentence	Individual
Estimable	A story does not denote a coarse-grained requirement that is difficult to plan and prioritize	Individual
Unique	Every user story is unique, duplicates are avoided	Set
Uniform	All user stories in a specification employ the same template	Set
Independent	The user story is self-contained and has no inherent dependencies on other stories	Set
Complete	Implementing a set of user stories creates a feature-complete application, no steps are missing	Set

Table 2: Quality User Story framework that defines 13 criteria for user story quality [35]

a database of points of interest that is not mentioned in any of the other stories. An important additional function that is misrepresented as an end, but should be a means in a separate US, for example:

- "As a user, I would like to open the interactive map";
- "As a user, I would like to see the location of points of interest on the interactive map".

problem orientated

According to the principle of problem specification for requirements engineering proposed by Zave and Jackson, a US should only specify the problem. If absolutely necessary, implementation notes can be included as comments or descriptions. Apart from the violation of the minimum quality criteria, this US contains "As a care professional, I would like to save a refund - Save button top right (never greyed out)" Implementation details (a solution) within the US text. The text could be rewritten as follows "As a carer I would like to save a reimbursement".

Unambiguous

Ambiguity is inherent in natural language requirements, but the requirements engineer writing USs must avoid it as much as possible. A US should not only be internally unambiguous, but should also be unambiguous in relation to all other USs. The taxonomy of ambiguity types [8] provides a comprehensive overview of the types of ambiguity that can occur in a systematic requirements specification.

In this US "As a user, I am able to edit the content I have added to a person's profile page", "content" is a superclass that refers to audio, video, and text media uploaded to the profile page, as specified in three other, separate USs in the real US record. The requester should explicitly mention which media is editable; for example, the story can be modified as follows: "As a user, I am able to edit video, photo and audio content that I have added to a person's profile page".

Full sentence

A US should read like a complete sentence, without typos or grammatical errors. The US "server configuration", for example, is not formulated as a complete sentence (and does not correspond to the syntactic quality). By reformulating the feature as a complete sentence US, it automatically specifies what exactly needs to be configured. For example, US "Server configuration" can be converted to "As an administrator, I would like to configure the sudo-ers of the server".

Estimatable

The larger and more complex the US becomes, the more difficult it is to accurately estimate the required effort. Therefore, any US should not become so large that it becomes impossible to estimate and plan with reasonable certainty[6].For example, the US "As a carer, I would like to see my route list for the next/future days so that I can prepare (e.g. I can see when I should start the journey)" a route list so that carers can prepare.

This may be an unordered list of places to visit during the working day. However, it is equally likely that the function includes an algorithmic arrangement of routes to minimise the distance travelled and/or display the route on a map. These many functionalities make an accurate estimate difficult and make it necessary to split the US into several USs, for example:

- "As a carer, I would like to see my route list for the next/future days so that I can prepare";
- "As a manager, I would like to upload a route list for carers".

The following quality criteria refer to a "set" of USs. These quality criteria are relevant for assessing the quality of the entire project specification, as they focus on the entirety of the project specification and not on the individual review of each individual story:

[6] http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Unique and conflict-free

The concept of unique USs, which emphasises the avoidance of semantic similarity or duplication within a project. For example, consider EP_a: "As a visitor, I can see a list of messages so I can stay up to date" and US_a: "As a visitor, I can see a list of messages so I can stay up to date". This situation can be improved by offering more specific messages, such as:

- US_{a1} "As a visitor, I am able to see the latest news;"
- US_{a2} "As a visitor I am able to see sports news"

It is also important to avoid conflicts between USs to ensure their quality. A requirements conflict occurs when two or more requirements cause an inconsistency[41, 44]. For example, consider the story US_b: "As a User, I am able to edit any landmark" contradicts the requirement that a user can edit any landmark (US_c: "As a User, I am able to delete only the landmarks that I added"), if we assume that edit is a general term that includes delete. US_b refers to any landmark, while US_c refers only to those that the user has added. One possible way to fix this is to change US_b: "As a user, I can edit the landmarks I have added". [35]

To recognise these types of relationships, each US part must be compared with the parts of the other USs using a combination of similarity measures that are either syntactic (e.g. Levenshtein distance) or semantic (e.g. using an ontology to determine synonyms). If the similarity exceeds a certain threshold, a human analyst must analyse the USs for possible conflicts and/or duplicates.

Definition 2.1. *A US μ is a 4-tuple $\mu = (r, m, E, f)$, where r is the role, m is the mean, $E = (e_1, e_2, ...)$ is a set of ends and f is the format. A means m is a 5-tuple $m(s, av, do, io, adj)$, where s is a subject, av an action verb, do a direct object, io an indirect object and adj an adjective (io and adj can be zero, see figure 2). The set of user stories in a project is denoted by $U = (\mu_1, \mu_2, ...)$.*

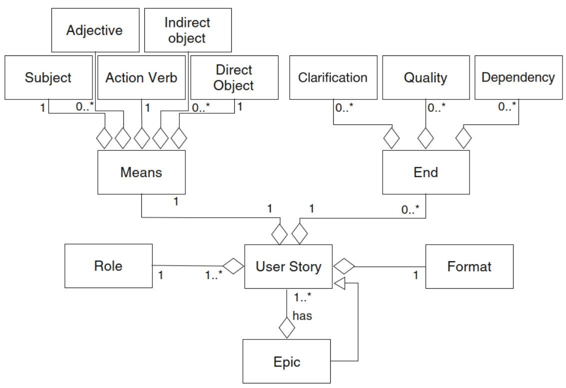


Figure 2: Conceptual model of the USs [35]

Uniform

Uniformity refers to the consistency of a US format where the majority of USs are within the same set. To assess uniformity, the requirements engineer determines the most common format, which is usually determined in collaboration with the team. For example, the US "As administrator, I receive an email notification when a new user is registered" is presented as a non-uniform US and can be rewritten as follows to improve uniformity: "As an administrator, I would like to receive an email notification when a new user is registered".

Independent

USs should be able to be planned and implemented in any order and should not overlap conceptually.

It is recommended that all dependencies are made explicit and visible, as complete independence is not always achievable. In addition, some interdependencies may not be possible to resolve, and you may want to consider making these interdependencies visible in a practical way, for example by adding notes to story cards or by linking to them in the issue tracker. Two examples of dependency are given:

- *causality*: Sometimes a US (l_1) needs to be completed before another (l_2) may start. which states that l_1 is causally dependent on l_2 if certain conditions are satisfied.
- *superclasses*: USs can contain an object (*e.g.*, "content" in US "As a user, I can edit the content I've added to a person's profile page") that references several other objects in different histories. This means that the object in l_1 serves as a parent or superclass for the other objects.

Complete

The implementation of a series of USs should result in a complete application. Whilst it is not necessary for the USs to cover 100% of the application's functionality from the outset, it is important not to overlook any essential USs as this can create a significant functionality gap that hinders progress. Take for example the US "As a user, I can edit the content I have added to a person's profile page", which requires the presence of another story describing the creation of content. This scenario can be extended to USs with action verbs that refer to non-existent direct objects, such as reading, updating or deleting an item, which requires its prior creation. To address these dependencies with respect to the direct object of the agent, Lucassen et al. introduce a conceptual relation.

The Automatic Quality User Story Artisan (AQUSA)

The QUS framework provides guidelines for improving the quality of USs. To support the framework, Lucassen et al. propose the AQUSA tool, which exposes defects and deviations from good US practice [35]. AQUSA primarily targets easily describable and algorithmically determinable defects in the clerical part of requirements engineering, focusing on syntactic and some pragmatic criteria, while omitting semantic criteria that require a deep understanding of requirements' content [35]. AQUSA consists of five main architectural components (Figure 3): linguistic parser, US base, analyzer, enhancer, and report generator.

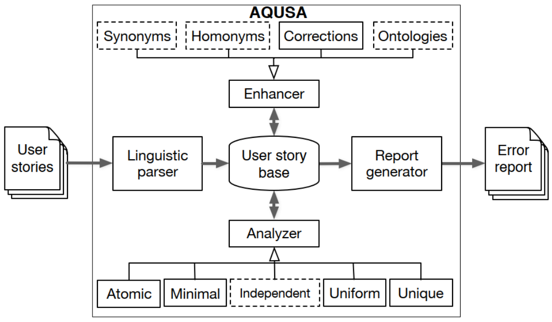


Figure 3: Functional view on architecture of AQUSA. Dashed components are not fully implemented yet [35]

The first step for every US is validating that it is well-formed. This takes place in the linguistic parser, which separates the US in its role, means and end(s) parts. The US base captures the parsed US as an object according to the conceptual model, which acts as central storage. Next, the analyzer runs tailormade method to verify specific syntactic and pragmatic quality criteria—where possible enhancers enrich the US base, improving the recall and precision of the analyzers. Finally, AQUSA captures the results in a comprehensive report [35].

In the case of story analysis, AQUSA v1 conducts multiple analyses, beginning with the *StoryChunker* and subsequently executing the Unique-, Minimal-, WellFormed-, Uniform-, and *AtomicAnalyzer* modules. If any of these modules detect a violation of quality criteria, they engage the *DefectGenerator* to record the defect in the associated database tables related to the story. Additionally, users have the option to utilize the AQUSA-GUI to access a project list or view a report of defects associated with a set of stories.

Linguistic Parser: Well-Formed

One of the essential aspects of this is the division of US into *role*, *means* and *end(s)*. This initial step is performed by the linguistic parser, implemented as the StoryChunker component. It identifies common indicators in the US, such as “As a”, “I want to”, “I am able to”, and “so that”. The linguistic parser then categorizes words within each chunk using the Stanford NLP POS Tagger and validates the following rules for each chunk:

- Role: Checks if the last word is a noun representing an actor and if the words preceding the noun match a known role format (*e.g.*, “as a”).

- Means: Verifies if the first word is “I” and if a known means format like “want to” is present. It also ensures the remaining text contains at least one verb and one noun (*e.g.*, “update event”).

- End: Checks for the presence of an end and if it starts with a recognized end format (*e.g.*, “so that”).

The linguistic parser validates whether a US adheres to the conceptual model. When it cannot detect a known means format, it retains the full US and eliminates the ”role” and ”end” sections. If the remaining text contains both a verb and a noun, it’s tagged as a “potential means,” and further analysis is conducted. Additionally, the parser checks for a comma after the role section.

User Story Base and Enhancer

Linguistically parsed USs are transformed into objects containing role, means, and ends components, aligning with the first level of decomposition in the conceptual model. These parsed USs are stored in the US base for further processing. AQUSA enriches these USs by adding potential synonyms, homonyms, and relevant semantic information sourced from an ontology to the pertinent words within each chunk. Additionally, AQUSA includes a corrections of subpart, ensuring precise defect correction where possible.

Analyzer: Explicit Dependencies

AQUSA enforces that USs with explicit dependencies on other USs should include navigable links to those dependencies. It checks for numbers within USs and verifies whether these numbers are enclosed within links. For instance, if a US reads, “As a care professional, I want to edit the planned task I selected—see 908”, AQUSA suggests changing the isolated number to “See PID-908,” where PID represents the project identifier. When integrated with an issue tracker like Jira or Pivotal Tracker, this change would automatically generate a link to the dependency, such as “see PID-908 (http://company.issuetracker.org/PID-908.” It’s worth noting that this explicit dependency analyzer has not been implemented in AQUSA v1 to ensure its universal applicability across various issue trackers.

Analyzer: Atomic

AQUSA examines USs to ensure that the means section focuses on a single feature. To do this, it parses the means section for occurrences of the conjunctions “and, &, +, or”. If AQUSA detects double feature requests in a US, it includes them in its report and suggests splitting the US into multiple ones. For example, a US like “As a User, I’m able to click a particular location from the map and thereby perform a search of landmarks associated with that latitude-longitude combination” would prompt a suggestion to split it into two USs: (1) “As a User, I want to click a location from the map” and (2) “As a User, I want to search landmarks associated with the lat-long combination of a location.”

AQUSA v1 verifies the role and means chunks for the presence of the conjunctions “and, &, +, or”. If any of these conjunctions are found, AQUSA checks whether the text on both sides of the conjunction conforms to the QUS criteria for valid roles or means. Only if these criteria are met, AQUSA records the text following the conjunction as an atomicity violation.

Analyzer: Minimal

AQUSA assesses the minimality of USs by examining the role and means of sections extracted during chunking and *well-formedness* verification. If AQUSA successfully extracts these sections, it checks for any additional text following specific punctuation marks such as dots, hyphens, semicolons, or other separators. For instance, in the US “As a care professional I want to see the registered hours of this week (split into products and activities). See: Mock-up from Alice NOTE: First create the overview screen—Then add validations,” AQUSA would flag all text following the first dot (“.”) as non-minimal. Additionally, any text enclosed within parentheses is also marked as non-minimal. AQUSA v1 employs two separate minimality checks using regular expressions. The first check searches for occurrences of special punctuation marks like “- , ? , . , *.” and marks any text following them as a minimality violation. The second check identifies text enclosed in brackets such as “(), [], {}, <>” and records it as a minimality violation.

Analyzer: Uniform

AQUSA, in addition to its chunking process, identifies and extracts the format parts of USs and calculates their occurrences across all USs in a set. The most frequently occurring format is designated as the standard US format. Any US that deviates from this standard format is marked as non-compliant and included in the error report. For example, if the standard format is “I want to,” AQUSA will flag a US like “As a User, I am able to delete a landmark” as non-compliant because it does not follow the standard. After the linguistic parser processes all USs in a set, AQUSA v1 initially identifies the most common US format by counting the occurrences of indicator phrases and selecting the most frequent one. Later, the uniformity analyzer calculates the edit distance between the format of each individual US chunk and the most common format for that chunk. If the edit distance exceeds a threshold of 3, AQUSA v1 records the entire story as a uniformity violation. This threshold ensures that minor differences, like “I am” versus “I’m,” do not trigger uniformity violations, while more significant differences in phrasing, such as “want” versus “can,” “need,” or “able,” do.

Analyzer: Unique

AQUSA has the capability to utilize various similarity measures, leveraging the WordNet lexical database to detect semantic similarity. For each verb and object found in the means or end of a US, AQUSA performs a WordNet::Similarity calculation with the corresponding verbs or objects from all other USs. These individual calculations are combined to produce a similarity degree for two USs. If this degree exceeds 90%, AQUSA flags the USs as potential duplicates.

AQUSA-GUI: report generator

After AQUSA detects a violation in the linguistic parser or one of the analyzers, it promptly creates a defect record in the database, including details such as the defect type, a highlight of where the defect is located within the US, and its severity. AQUSA utilizes this data to generate a comprehensive report for the user. The report begins with a dashboard that provides a quick overview of the US set's quality. It displays the total number of issues, categorized into defects and warnings, along with the count of perfect stories. Below the dashboard, all USs containing issues are listed, accompanied by their respective warnings and errors. An example is illustrated in figure 4.

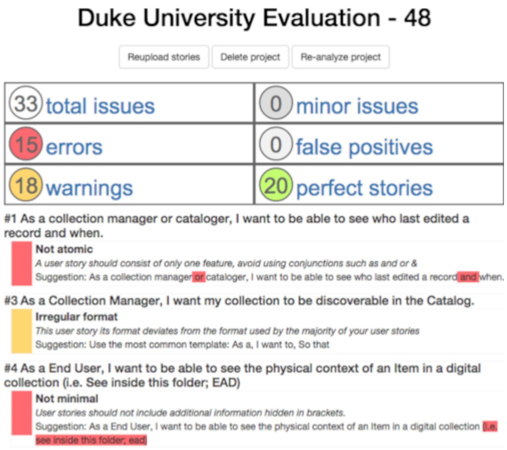


Figure 4: Example report of a defect and warning for a story in AQUSA [35]

Conclusion

In the QUS framework, a conflict is defined as a requirements conflict that occurs when two or more requirements cause an inconsistency. As far as the inconsistency is concerned, it is not clear to which form it actually belongs. Is it a content inconsistency or an inconsistency in the sense that two USs are very close to each other and describe the matter slightly differently, which leads to an inconsistency that we can understand as a similarity?

On the one hand, there may be conflicts between the finalised system and the inconsistencies in the specification of the USs. The content-related conflict in the quality criterion is not even mentioned. They described how the USs were written down and do not refer to content conflicts.

Both conflicts and redundancy are interesting and both terms and both analyses we would sharpen and make clear what can be analysed at all. However, this only partially fits in with the quality framework here from AQUSA-tool.

Example 2.1. *Considering two USs:*
US_1*: "As a user, I am able to edit any landmark" and* US_2*: "As a user, I am able to delete only the landmarks that I added". First, we try to minimize* US_1 *and divided it into three USs namely:*

- US_1a*:"As a user, I am able to add any landmark."*
- US_1b*:"As a user, I am able to modify any landmark."*
- US_1c*:"As a user, I am able to delete any landmark."*

US_1c *means that two users are allowed to delete the same landmark, which would lead to a conflict. This conflict can be avoided if* US_1c *is replaced with* US_2*, as two users are then no longer allowed to delete the same landmark. Furthermore, this situation cause an inconsistency between* US_1c *and* US_2*, e.g. if* US_2 *deletes the landmark that was added by a particular user,* US_1c *can no longer find this landmark and vice versa.*

The tool AQUSA can identify exact duplicates of USs or similarities. However, it lacks a more advanced uniqueness check that fully considers the conceptual model of USs. In this context, we would like to contribute by addressing this unmet need.

3 Preliminaries

In the 3.1 Section, we introduce *conditional random fields* (CRF) for the extraction of domain models from agile product backlogs, which play a central role in effectively supporting the identification of dependencies and conflicts between user stories. Furthermore, we conduct a conclusion.

Next, we dive into the Section 3.2 and introduce *natural language processing* (NLP) and *VerbNet* as a computational lexicon resource as well as a conclusion.

Finally, in Section 3.3, some basic definitions of *graphs* and *graph transformation rules* are laid down for better understanding. We then look at the graph transformation tool *Henshin* and its extension *conflict and dependency analysis* (CDA), which plays a central role in redundancy analysis. We also draw a conclusion and explain why we have chosen these techniques and what their central idea is.

3.1 Extracting Domain Models from Textual Requirements

A domain model is a conceptual representation of the entities, attributes and relationships within a US. It serves as an abstract blueprint for understanding and developing a system by capturing the essential aspects of the domain it represents.

Automated support for extracting domain models from requirements artefacts such as USs play a central role in effectively supporting the detection of redundancies and conflicts between USs. In this section, we present a graph-based extraction modelling with Doccano tool presented by Mosser et al. and conclude our review.

A Modelling Backlog as Composable Graphs

Mosser et al. propose a model engineering method (and the associated tooling) to exploit a graph-based meta-modelling and compositional approach. The objective is to shorten the feedback loop between developers and POs while supporting agile development's iterative and incremental nature.

The tool can extract what is called a conceptual model of a backlog in an ontology-like way. The conceptual models are then used to measure USs quality by detecting ambiguities or defects in a given story [40]. From a modelling point of view, Mosser et al. represents the concepts involved in the definition of a backlog in a metamodel, as depicted in figure 5. Without surprise, the key concept is the notion of story, which brings a benefit to a *Persona* thanks to an *Action* performed on an *Entity*. A Story is associated to a readiness *Status*, and might optionally contribute to one or more *QualityProperty* (*e.g.*, security, performance).

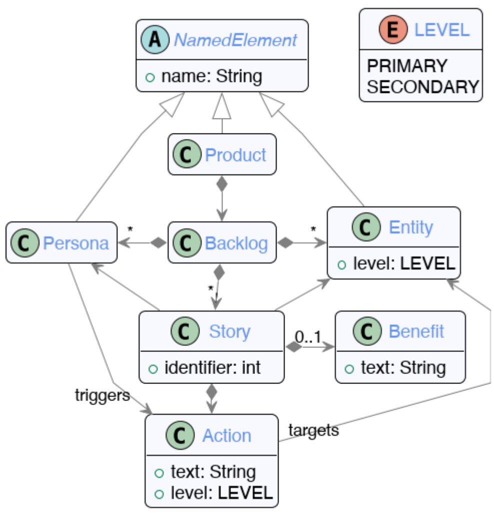


Figure 5: Backlog conceptual metamodel [40]

Consider, for example, the following story, extracted from the reference dataset [13]: "As a user, I want to click on the address so that it takes me to a new tab with Google

Maps.". *This story brings to the user (Persona) the benefit of reaching a new Google Maps tab (Benefit) by clicking (Action) on the displayed address (Entity).*

As entities and personas implement the *jargon* to be used while specifying features in the backlog, they are defined at the *Backlog level*. On the contrary, Actions belong to the associated stories and are not shared with other stories. Finally, a *Product* is defined as the *Backlog* used to specify its features.

Mosser et al. propose a system for use in the context of backlog management, which is shown in Figure 6. It builds on the efficiency of NLP approaches. Mosser et al. suggest employing an NLP-based extractor to create a backlog model. This model will subsequently assist teams in the planning phase by aiding in the selection of stories for implementation during the upcoming iteration [40].

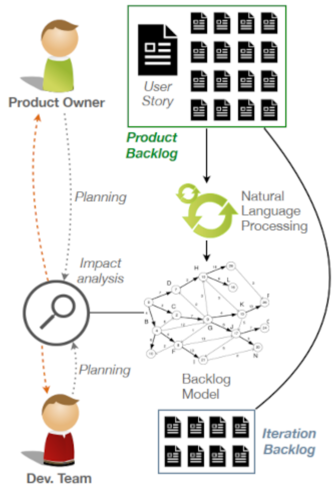


Figure 6: Providing early feedback at the backlog level [40]

Composable Backlogs

In order to support team customization (*e.g.*, a given team might want to enrich the backlog metamodel with additional information existing in their product management system), Mossser et al. chose open-world (ontological) representation by modelling backlog as graphs [40]. The graph is equipped with constraints (*e.g.*, a story always refers to a persona and an entity) to ensure that the minimal structure captured in the previously defined metamodel is guaranteed.

Definition 3.1 (**Story**). *A Story* $s \in S$ *is defined as a tuple* (P, A, E, K)*, where* $P = \{p_1, ..., p_i\}$ *is the set of involved personas,* $A = \{a_1, ..., a_i\}$ *the set of performed actions, and* $E = \{e_1, ..., e_k\}$ *the set of targeted entities. Additional knowledge (e.g., benefit, architectural properties, status) can be declared as key-value pairs in* $K = \{(k_1, v_1), ..., (k_l, v_l)\}$*. The associated semantics is that the declared actions bind personas to entities. Considering that story independence is a pillar of agile methods (as, by definition, stories are* *independent inside a backlog), there is no equivalence class defined over* $S : \forall(s, s') \in S^2, s \neq s' \Rightarrow s \not\equiv s'$.

Mosser et al. suggest that USs are treated as if they were independent units within a backlog. In practice, however, this is not the case. As the backlog grows and different stakeholders are involved, redundancies and conflicts between them are inevitable.

Definition 3.2 (Backlog). *A backlog* $b \in B$ *is represented as an attributed typed graph* $b = (V, E, A)$*, with* V *a set of typed vertices,* E *a set of undirected edges linking existing vertices, and* A *a set of key-value attributes. Vertices are typed according to the model element they represent* $v \in V, type(v) \in \{Persona, Entity, Story\})$ *. Edges are typed according to the kind of model elements they are binding. Like backlogs, vertices and edges can contain attributes, represented as* (key, value) *pairs. The empty backlog is denoted as* $\emptyset = (\emptyset, \emptyset, \emptyset)$.

Example 3.1. *Backlog excerpt: Content Management System for Cornell University — CulRepo* [13].

- s_1*: As a faculty member, I want to access a collection within the repository.*

Associated model:

- $s_1 = (\{facultymember\}, \{access\}, \{repository, collection\}, \emptyset) \in S$

A backlog containing a single story s_1*: ("As a faculty member, I want to access a collection within the repository").*

$$
\begin{aligned}
b_1 &= (V_1, E_1, \emptyset) \in B \\
V_1 &= \{Persona(faculty\ member, \emptyset)\ , \\
&\quad Stroy\,(s_1, \{(action, access)\}) \\
&\quad Entity\,(repository, \emptyset), \\
&\quad Entity(collection, \emptyset)\} \\
E_1 &= \{has\_for\_persona(s_1, faculty\ member), \\
&\quad has\_for\_entity\,(s_1, repository) \\
&\quad has\_for\_entity(s_1, collection)\}
\end{aligned}
$$

Conditional Random Fields (CRF)

CRFs [29] are a particular class of *Markov Random Fields*, a statistical modelling approach supporting the definition of discriminative models. They are classically used in pattern recognition tasks (labelling or parsing) when context is important identify such patterns [6].

To apply CRF Mosser et al. transform a given story into a sequence of tuples. Each tuple contains minimally three elements: *(i)* the original word from the story, *(ii)* its syntactical role in the story, and *(iii)* its semantical role in the story. The syntactical role in the sentence is classically known as *Part-of-Speech* (POS), describing the grammatical role of the word in the sentence. The semantical role plays a dual role here. For training the model, the tags will be extracted from the annotated dataset and used as target. When used as a predictor after training, these are the data Mosser et al. will ask the model for infer.

The main limitations of CRF are that *(i)* it works at the word level (model elements can spread across several words), and *(ii)* it is not designed to identify relations between entities [6]. To address the first limitation, Mosser et al. use a glueing heuristic. Words that are consecutively associated with the same label are considered as being the same model element, *e.g.*, the subsequence ["UI", "designer"] from the previous example is considered as one single model element of type *Persona*.

Mosser et al. applied this heuristic to everything but verbs, as classically, two verbs following each other represent different actions. They used again heuristic approach to address the second limitation. Mosser et al. bound every *Persona* to every primary *Action* (as *trigger* relations), and every primary *Actions* to every primary *Entity* (as *target* relations) [6].

Example 3.2. *Consider the following example:*

$S = ['As', 'a', 'UI', 'designer', ',', \ldots]$
$POS(S) = [ADP, DET, NOUN, NOUN, PUNCT, \ldots]$
$Label\,(S) = [\emptyset, \emptyset, PERSONA, PERSONA, \emptyset, \ldots]$

S represents a given US (Table 6). POS (S) represent the Part-of-speech analysis of S. The story starts with an adposition (ADP), followed by determiner (DET), followed by a noun, followed by another noun, ... Then, Lables (S) represents what we interest in: the first two words are not interesting, but the 3^{rd} *and* 4^{td} *words represent a Persona.* A complete version of the example is provided in Table 6.

Word	As	a	UI	designer	,	I	want	to	begin	user	testing	,
POS	ADP	DET	NOUN	NOUN	PUNCT	PRON	VERB	PART	VERB	NOUN	NOUN	PUNCT
Label	-	-	PER	PER	-	-	-	-	P-ACT	P-ENT	P-ENT	-

Word	so	that	I	can	validate	stakeholder	UI	improvement	requests	.
POS	SCONJ	SCONJ	PRON	AUX	VERB	NOUN	NOUN	NOUN	NOUN	PUNCT
Label	-	-	-	-	S-ACT	S-ENT	S-ENT	S-ENT	S-ENT	-

POS tags are the Universal POS tags
Labels: PER (Persona), P-ACT (Primary Action), P-ENT (Primary Entity), S-ACT (Secondary Action), S-ENT (Secondary Entity)

Table 6: Minimal Feature Set, associating part-of-speech (POS) and semantic labels to each word in a given story [6]

Doccano Tool

Doccano[7] is an open-source annotation tool designed to facilitate the labelling and tagging of textual data for various NLP tasks. Its user-friendly interface and versatile functionalities make it suitable for different annotation purposes, including named entity recognition (NER), text classification, and sequence labelling. It supports collaborative annotation efforts, making it an ideal choice for projects requiring high-quality labelled datasets.

Doccano proved to be an effective tool for Mosser et al. in annotating USs, enabling them to create a reliable dataset with high inter-rater agreement. Its ease of use, collaborative features, and support for complex annotation schemes make it a valuable asset for any annotation project in the field of NLP.

Annotating User Stories with Doccano

Mosser et al. used publicly available requirements, focusing on a dataset from Dalpiaz et al.[13], which includes 22 product backlogs and 1,679 USs. This dataset is a raw archive of 22 text files, each containing USs for a product, one per line. They noted the absence of publicly available expert annotations providing ground truth for the dataset's concepts.

To address this, they manually annotated the dataset using the Doccano tool as a NER task. They applied labels such as persona, action, entity, and benefit part, along with three relations (targets, triggers, and contains) from their domain meta-model. Figure 7 shows example of annotated US using Doccano. Quality checks included:

- Initial calibration on 75 randomly selected stories.
- Fortnightly validation sessions over two months.
- Manual review of 330 randomly selected annotated stories (19.6%).

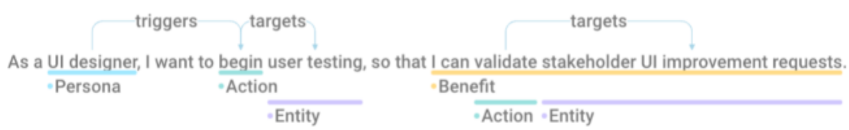


Figure 7: Example of annotated user using Doccano Annotation UI [6]

They found no significant discrepancies, with only 21 elements disputed, resulting in a 94% inter-rater agreement, considered excellent.

[7] https://doccano.github.io/doccano

Conclusion

The CRF approach is graph-based and is designed to achieve a high degree of precision and recall, which is particularly important in the context of domain concept extraction. The CRF approach can encompass both syntactic and semantic aspects, particularly when complemented by a suitable conceptual meta-model.

However, due to the aforementioned limitations of the CRF tool, specifically the exclusion of secondary actions/entities within relations such as triggers and targets, we opted for annotated datasets using Doccano presented by Mosser et al. This was because it labelled a greater number of relations such contains and targets, even between secondary actions/entities, which should facilitate precise evaluation in further analysis.

The datasets annotated with the Doccano tool presented by Mosser et al. can then serve as primary input for analysing conflicts and redundancies.

3.2 NLP and VerbNet as a Computational Lexical Resource

NLP is a computational method for the automated analysis and representation of human language [10]. NLP techniques offer potential advantages to improve the quality of USs and can be used to parse, extract, or analyze US's data. It has been widely used to help in the software engineering domain (*e.g.*, managing software requirements [5], extraction of actors and actions in requirement document [3].

NLP techniques are usually used for text preprocessing (*e.g.*, tokenization, *Part-of-Speech* (POS) tagging, and dependency parsing). Several NLP approaches can be used *e.g.*, syntactic representation of text and computational models based on semantic features. Syntactic methods focus on word-level approaches, while the semantic focus on multiword expressions [10].

A computational lexicon resource is a systematically organized repository of words or terms, complete with linguistic and semantic data. These lexicons play a pivotal role in facilitating NLP systems focused on semantic analysis by offering comprehensive insights into language elements, encompassing word forms, POS categories, phonetic details, syntactic properties, semantic attributes, and frequency statistics.

Lexical classes, defined in terms of shared meaning components and similar (morpho-)syntactic behaviour of words, have attracted considerable interest in NLP [10]. These classes are useful for their ability to capture generalizations about a range of (cross-)linguistic properties. NLP systems can benefit from lexical classes in a number of ways. As the classes can capture higher level abstractions (*e.g.* syntactic or semantic features) they can be used as a principled means to abstract away from individual words when required. Their predictive power can help compensate for lack of sufficient data fully exemplifying the behaviour of relevant words [27].

After completing the annotation of USs using the Doccano approach, where entities, actions (both primary and secondary), persona and their relational attributes (especially triggers, targets and contains) are carefully annotated and structured in the form of a graph-based representation, an initial imperative emerges. This imperative includes the determination of a representative semantic interpretation for the identified actions. This determination in turn serves as a prerequisite for the creation of the corresponding action annotations, namely the rules “create”, “delete”, “preserve” and “forbid”.

Conflict analysis depends on the application of a computational lexical resource technique, in particular VerbNet. This technique play a central role in providing the basic cognitive infrastructure that enables a comprehensive understanding of semantic roles and the systematic categorisation of linguistic elements, especially verbs embedded in the construct of US into ‘create’, ‘delete’, ‘preserve’ and, ‘forbid’.

VerbNet

VerbNet (VN) is a hierarchical domain-independent, broad-coverage verb lexicon with mappings to several widely-used verb resources, including WordNet [38], Xtag [42], and FrameNet [7]. It includes syntactic and semantic information for classes of English verbs derived from Levin’s classification, which is considerably more detailed than that included in the original classification.

Each verb class in VN is completely described by a set of members, thematic roles for the predicate-argument structure of these members, selectional restrictions on the arguments, and frames consisting of a syntactic description and semantic predicates with a temporal function, in a manner similar to the event decomposition of Moens and Steedman [39]. The original Levin classes have been refined, and new subclasses added to achieve syntactic and semantic coherence among members.

The VerbNet Hierarchy

VerbNet represents a hierarchical structure of verb behaviour, with groups of verb classes sharing similar semantics and syntax. Verb classes are numbered based on common semantics and syntax, and classes with the same top-level number (e.g., 9-109) have corresponding semantic relationships.

For instance, classes related to actions like “putting”, such as “put-9.1”, “put_spatial-9.2”, “funnel-9.3”, all belong to class number 9 and relate to moving an entity to a location. Classes sharing a top class can be further divided into subclasses, as seen with “wipe” verbs categorized into “wipe_manner” (10.4.1) and “wipe_inst” (10.4.2) specifying the manner and instrument of “wipe” verbs in the “Verbs of Removing” group of classes (class number 10).

The classification encompasses class numbers 1-57, derived from Levin’s classification [31], and class numbers 58-109, developed later by Korhonen and Briscoe [28]. The later classes are more specific, often having a one-to-one relationship between verb type and verb class. This hierarchical structure helps categorize and organize verbs based on their semantic and syntactic properties.

Verb Class Hierarchy Contents

Each individual verb class within VerbNet is hierarchical. These classes can include one or more “subclasses” or “child” classes, as well as “sister” classes. All verb classes have a top-level classification, but some provide further specification of the behaviours of their verb members by having one or more subclasses.

Subclasses are identified by a dash followed by a number after the class information. For example, the top class might be "spray-9.7", and a subclass would be denoted as "spray 9.7-1". This hierarchy allows for a more detailed and structured organization of verb behaviour within VerbNet.

- Top Class: The highest class in the hierarchy; all features in the top class are shared by every verb in the class. The top class of the hierarchy consists of syntactic constructions and semantic role labels that are shared by all verbs in this class.
- Parent Class: Dominates a subclass; all features are shared with subordinate classes.
- Subclasses: VerbNet subclasses inherit features from the top class but specify further syntactic and semantic commonalities among their verb members. These can include additional syntactic constructions, further selectional restrictions on semantic role labels, or new semantic role labels.
- Child Class: Is dominated by a parent class; inherits features from this parent class, but also adds information in the form of additional syntactic frames, thematic roles, or restrictions.
- Sister Class: A subclass directly dominated by a parent class. This parent class also, directly dominates another subclass, so the two subclasses are sisters to one another. Sister classes do not share features.

Figure 8 illustrate an example of class hierarchy from spray-9.7 class. Verb classes are

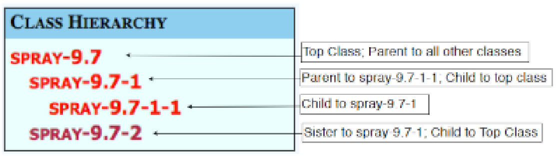


Figure 8: Class hierarchy for spray-9.7 class [21]

numbered according to shared semantics and syntax, and classes which share a top-level number (9-109) have corresponding semantic relationships.

For instance, verb classes related to putting, such as put-9.1, put_spatial-9.2, funnel-9.3, etc. are all assigned to the class number 9 and related to moving an entity to a location.

An example of top-class numbers[8] and their corresponding types is given in Table 8.

[8]https://verbs.colorado.edu/verb-index/VerbNet_Guidelines.pdf

Class Number	Verb Type	Verb Class
10	Verbs of Removing	banish-10.2 cheat-10.6.1 clear-10.3 debone-10.8 fire-10.10 mine-10.9 pit-10.7 remove-10.1 resign-10.11 steal-10.5 wipe_manner-10.4.1
26	Verbs of Creation and Transformation	adjust-26.9 build-26.1 convert-26.6.2 create-26.4 grow-26.2.1 knead-26.5 performance-26.7 rehearse-26.8 turn-26.6.1
13	Verbs of Change of Possession	berry-13.7 contribute-13.2 equip-13.4.2 exchange-13.6.1 fulfilling-13.4.1 future_having-13.3 get-13.5.1 give-13.1 hire-13.5.3 obtain-13.5.2

Table 8: An example of top-class numbers and their corresponding verb-type

Conclusion

The methodical semantic categorisation of verbs into hierarchical superclasses in VerbNet provides a structured and all-encompassing framework for understanding verb behaviour.

This matching is necessary for conflict analysis as it helps with the semantic interpretation of the actions (verbs) identified by the Doccano tool. The use of VerbNet's superclasses speeds up the annotation process and allows us to categorise not only individual verbs but also groups of verbs with the same semantic meaning into four categories (create, delete, preserve or forbid).

3.3 Analysis by Graph Transformation Tool

In a software development process, the class architecture is getting changed over the development, *e.g.* due to a change of requirements, which results in a change of the class diagram. During runtime of a software, an object diagram can also be modified trough creating or deleting of new objects.

Many structures, that can be represented as graph, are able to change or mutate. This suggests the introduction of a method to modify graphs through the creation or deletion of nodes and edges. This graph modification can be performed by the so-called graph transformations. There are many approaches to model *graph transformations e.g.* the double push-out approach or the single push-out approach, which are both concepts based on push-outs from category theory in the category Graphs.

In this section, we first present some basic definitions of graph transformation and transformation rules for a better understanding. We then discuss the graph transformation tool Henshin and its CDA tool [4], which play a central role in our methodology.

Graphs and Typed Graphs

A graph consists of nodes and edges, with each edge connecting precisely two nodes and having the option to be directed or undirected. When an edge is directed, it designates a distinct start node (source) and an end node (target). For the purpose of this discussion, we will focus on directed graphs.

Definition 3.3 (**graph**). *A* graph $G = (V, E, s, t)$ *contains* V*, a set of nodes,* E*, a set of edges,* $s : E \to V$*, a source function, where* $s(e)$ *is the start node of* $e \in E$ *and a target function* $t : E \to V$*, where* $t(e)$ *is the end node of a edge* $e \in E$*.*

Definition 3.4 (**Transformation Rule**). *A* transformation rule *denotes which nodes and edges of a graph have to be deleted and which nodes and edges have to be created. In the double-pushout approach a transformation rule* $p = L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R$ *consists of three graphs* L, K, R*, two graph morphisms* $l : K \to L$ *and* $r : K \to R$*, where* K *contains all elements, that remain in the graph,* $L \setminus l(K)$ *contains the elements that are removed and* $R \setminus r(K)$ *contains the elements, that are created.*

Definition 3.5 (**Graph Transformation**). *In the context of* graph transformations*, when we have two graphs* G *and* H*, along with a transformation rule* p*, we can apply this rule to graph* G *at match* m*. This application, denoted as* $G \overset{p,m}{\Longrightarrow} H$*, results in graph* H*. The match, represented as* $m : L \hookrightarrow G$*, is an injective graph morphism, and* L *contains all the nodes and edges of* p *that remain intact and are not deleted during the transformation.*

As described in Section 3.1, where we explained our use of the Doccano tool as a graph-based metamodelling tool to annotate USs, each US is carefully structured and annotated in the form of a graph. We then apply transformation rules to these Doccano-generated graphs to generating graph transformation rules using Henshin API for analysing redundancies between USs using CDA tool.

Henshin: A Tools for In-Place EMF Model Transformations

The Eclipse Modelling Framework (EMF) provides modelling and code generation facilities for Java applications based on structured data models. Henshin is a language and associated tool set for in-place transformations of EMF models.

The Henshin transformation language uses pattern-based rules on the lowest level, which can be structured into nested transformation units with well-defined operational semantics. So-called amalgamation units are a special type of transformation units that provide a forall-operator for pattern replacement. For all these concepts, Henshin provides a visual syntax, sophisticated editing functions, execution and analysis tools. The Henshin transformation language has its roots in attributed graph transformations, which offer a formal foundation for validation of EMF model transformations [4].

Graph Types

Graph transformation-based approaches, essentially define model transformations using rules consisting of a pre-condition graph, called the left-hand side (LHS), and a post-condition graph, called the right-hand side (RHS) of the rule. Informally, the execution of a model transformation requires that a matching of objects in the model (host graph) to the nodes and edges in the LHS is found, and these matched objects are changed in such a way that the nodes and edges of the RHS match these objects [49].

The performance of graph transformation-based model is mainly determined by the efficiency of the match finding of the LHS. Consequently, model transformation languages offer different options to add constraints to the LHS of model transformations to improve the performance of the matching [49]. To be efficient, graph transformation tools usually employ heuristics such as search plans to provide good performance (e.g. [50]).

Structure and Application of Rules

The Henshin transformation language is defined by means of a metamodel. The Henshin metamodel is closely aligned to the underlying formal model of double pushout (DPO) graph transformations [49]. Thus, rules consist of a left-hand side and a right-hand side graph as instances of the *Graph* class. Rules further contain node mappings between the LHS and the RHS which are omitted here for better readability. Graphs consist of a set of Nodes and a set of Edges. Nodes can additionally contain a set of Attributes. These three kinds of model elements are typed by their corresponding concepts in the Ecore metamodel of EMF.

Application Conditions

To conveniently determine where a specified rule should be applied, application conditions can be defined. An important subset of application conditions is negative application conditions (NACs) which specify the non-existence of model patterns in certain contexts. In the Henshin transformation model, graphs can be annotated with application conditions using a *Formula*. This formula is either a logical expression or an application condition, which is an extension of the original graph structure by additional nodes and edges. A rule can be applied to a host graph only if all application conditions are fulfilled [4].

Attribute and Parameters

Nodes may also include a set of *Attributes*. *Rules* inherit from *Units* and can thus include various *Parameters*. A common use of parameters is to transmit an attribute value (such as a name) from a node to be matched in the rule. To restrict the application of a rule, the metamodel encompasses concepts for representing nested graph conditions [18] as well as attribute conditions.

State Space Exploration

Henshin support in-place model transformation, Arendt et al. have developed a state space generation tool, which allow to simulating all possible executions of a transformation for a given input model, and to apply model checking, similar to the GROOVE [23] tool. Henshin can generate finite as well as large state space exploration. Regarding generation and analysis of large state space, the tool's ability to utilize parallel algorithms, taking advantage of modern multi-core processors, which enables the handling of state spaces with millions of states.

Analysing Conflicts

The conflict and dependency analysis (CDA) for graph rewriting [19] has been adapted to rule-based model transformation, *e.g.* to find conflicting functional requirements for software systems [20]. The application of a rule r_1 is in conflict with the application of a rule r_2 if

- r_1 deletes a model element used by the application of r_2 **(delete/use)**, or
- r_1 produces a model element that r_2 forbids **(produce/forbid)**, or
- r_1 changes an attribute value used by r_2 **(change/use)**.

Different between Model Checking and Conflict and Dependency Analysis

In this section, we shall delineate two distinct analytical methodologies, specifically model checking and conflict and dependency analysis. Their respective purposes are delineated in Table 10, which serves to elucidate their appropriateness for modelling USs.

Example 3.3. *Considering US_1: "As a user, I am able to edit any landmark." and US_2: "As a user, I am able to delete only the landmarks that I added.". First, we minimize US_1 and divided it into three USs as follows:*

- *US_1a: "As a user, I am able to add any landmark."*
- *US_1b: "As a user, I am able to modify any landmark."*
- *US_1c: "As a user, I am able to delete any landmark."*

Aspect	Model Checking for User Stories	Conflict and Dependency Analysis for User Stories
Purpose	Verify user story properties and system behaviour	Understand dependencies and interactions between user stories
Method	Large state spaces exploration	Rule-based model transformation
Automated vs. Manual	Automated	Automated
Scope	Ensuring user stories meet specified requirements and system behaviour	Understanding how user stories relate to each other, managing dependencies
Use Cases	Ensuring user story correctness and system behavior	Agile development, impact analysis, and managing user story dependencies
Result	Verification of user story properties (e.g., acceptance criteria)	Identification of user story dependencies, potential conflicts, and their impact on the development process

Table 10: Comparative analysis between model checking and conflict and dependency methods

Figure 9 shows the class model "Map" while figure 10 shows the defined rules in Henshin, with each rule corresponding to a US.

In this example, we assume that a user only performs one action at a time. If US_1c is translated into a rule and then CDA is applied (Figure 11), Henshin would find a "Delete-Delete" conflict between two "Actions" (verbs) in US_1c and US_2 where two users are allowed to delete the same landmark. This conflict can be avoided if US_1c is replaced with US_2, as two users are then no longer allowed to delete the same landmark.

Figure 11 shows the conflicts between two rules reported by CDA tool. For Instance, there is a "Create-Delete" conflict between US_1b and US_2. If the specific landmark are deleted, US_1b cannot modify that landmark at all.

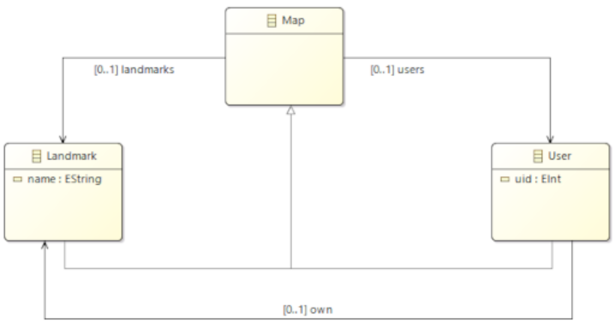


Figure 9: Henshin Class Model Map

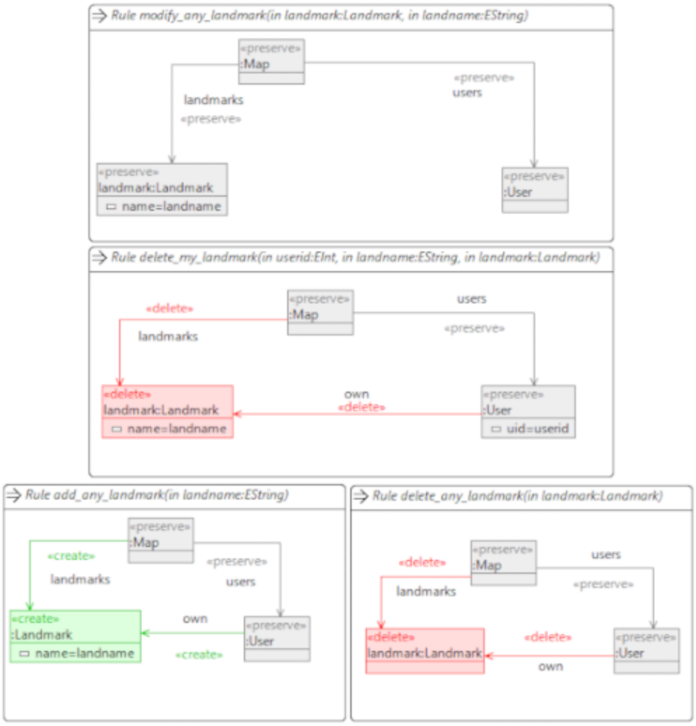


Figure 10: Illustrated rules in Henshin: add_any_landmark rule related to US_1a, modify_any_landmark rule related to US_1b, delete_any_landmark rule related to US_1c and delete_my_landmark rule related to US_2

CDA Tool

The provided CDA extension of Henshin can be used in two different ways: Its application programming interface (API) can be used to integrate the CDA into other tools and a user interface (UI) is provided supporting domain experts in developing rules by using the CDA interactively [37].

After invoking the analysis, the rule set and the kind of critical pairs to be analysed have to be specified. Furthermore, options can be customized to stop the calculation after finding a first critical pair, to ignore critical pairs of the same rules, etc. The resulting list of critical pairs is shown and ordered along rule pairs.

Conclusion

In our study, Henshin and the CDA tools play an important role in redundancy (and possibly conflict) analysis. They are the only graph transformation based techniques that provide us with comprehensive information about possible redundancies (and conflicts) between the elements of US-pairs.

Since Henshin enables the specification of constraints and conditions within rules,

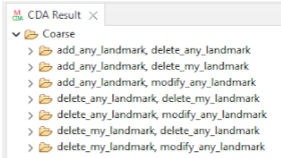


Figure 11: Henshin CDA result visualises conflicts between USs

which can be useful for enforcing and verifying US requirements to ensure that constraints are met.

Moreover, a compelling factor in favour of Henshin is the inherent support of a versatile *application programming interface* (API), facilitating seamless integration of Henshin functionality into various tools, including Java-based platforms.

4 Analysing Redundancies

USs as a fundamental component of agile software development provide concise and clear descriptions of the software functionality from the end user's perspective. These stories guide the development process and ensure that the software meets the actual needs of the users.

However, as the project backlog grows, they often become overloaded with redundant USs that can obscure project priorities, waste resources and complicate maintenance. Analysing the redundancy within these USs is key to maintaining clarity and efficiency in the development process.

The main goal of this analysis is to streamline the software development workflow by identifying redundancies in the USs within a project's backlog in a syntactic way.

By identifying and consolidating overlapping functionalities, companies can avoid redundant work, streamline development work and optimise testing processes. This strategic approach not only shortens project timelines but also improves the clarity and coherence of project documentation, making it easier for team members to navigate and manage project requirements.

In Section 4.1 we briefly discuss related work. In Section 4.2 we present the requirements and functional needs that are used as input for the design phase to fulfil them. In Section 4.3 we explain the design decisions of the workflow shown in Figure 12 and explain how the architecture is structured.

In Section 4.4 follows the implementation to show how the components and their classes look in relation to Figure 13. In Section 4.5 we show what and how far we tested it. Finally, we apply our approach to 19 datasets of backlogs annotated with the Doccano tool [9]. In Section 4.6 we analyse the results and in Section 4.7 we draw a conclusion.

[9] https://github.com/ace-design/nlp-stories

4.1 Related work

Existing methods for assessing the similarity and uniqueness of USs have largely relied on NLP techniques. These techniques have explored the concept of similarity in USs, recognizing that even when USs appear similar, they may not be redundant. For instance, consider the following examples: “As a Staff member, I want to Remove a Hold.” and “As a Staff member, I want to apply a Hold.” While these USs are similar, their actions differ, indicating that they are not redundant. Jurisch et al. utilized common NLP methods like k-means clustering and TF-IDF to group similar USs[15]. These NLP techniques often misidentify US pairs as redundant when they rely solely on similarity measures [15, 22, 31].

Lucassen et al. developed a conceptual model for USs that defines various measures of similarity and redundancy. They identify two main types of duplicates: full duplicates, where the USs are identical in wording, and semantic duplicates, where the USs convey the same meaning but are phrased differently. Additionally, they consider cases where two USs share the same benefit but have different main parts, or vice versa, and instances where the benefit of one US is the same as the main part of another, indicating a dependency between them. Even when the personas differ, the main parts or benefits can be the same, suggesting that a more general persona might be needed to unify these USs[36].

Lucassen et al. demonstrated that annotating USs can help identify similarities between their parts. Their tool, AQUSA, employs the WordNet::Similarity calculation to detect potential duplicates among USs. However, only two instances of redundancies are reported in their study, without details on the nature of these violations. Additionally, the evaluation data is not published, making it challenging to assess AQUSA's effectiveness in analysing the redundancy.

In our approach, we do not focus on the similarity between USs, but on the redundancies between USs using annotated USs and their individual elements.

4.2 Requirements

In order to accomplish the analysis of redundancies in USs we try to address following functional requirements:

- As a user, I want to perform syntactic analysis on user stories within a specified project backlog, so that I can identify and address redundancies effectively.
- As a User, I want a report of user story pairs that contain identical syntactic clauses in both the main and benefit parts, so that I can modify them as needed.
- As a User, I want to apply a filter to the redundancy report to exclusively display US-pairs, which have at least one clauses as *Targets* relation with the terms "Action" (as a verb) and "Entity" (as a noun), so that I can efficiently reduce the number of potential redundant pairs.

- As a user, I want to mark found redundancy clauses as Triggers relation with a hash symbol (#) and show those that have a redundancy in “Persona” (as a noun) and “Action” (as a verb) entries, so that I can better see if the persona is also recognised as a redundancy.

- As a user, I want to mark founded redundancy clauses as Contains with a hash symbol (#) and show those that contain two “Entity” (as a noun), so that I can better see whether the containments entities is also recognised as a redundancy.

- As a user, I would like to have a redundancy report that shows founded US texts in US-pairs and adds a hash symbol (#) at the beginning and end of the founded words as a marker, so that I can better see which words are redundant in US-pairs.

- As a user, I want to see how many redundancy clauses were founded in main and benefit parts of each US-pair, so that I can aggregate each founded redundancy US-pair for further statistical purposes on that basis.

- As a user, I want a table at the top of the redundancy report that lists the US-pairs and the number of redundancy clauses contained in each pair, so that I can quickly see all the US-pairs founded and the number of redundancy clauses.

- As a user, I want to know whether the redundancy clauses belong to the main or benefit part of the US, so that I can use it for further statistical purposes.

- As a user, I want to know if a pair of US is partially or fully redundant, so that I can delete or modify them.

To judge the operation of a system, we define following non-functional requirements:

- Testability: The system should support automated test procedures to ensure that syntactic analysis and redundancy detection work correctly. It should include comprehensive test cases covering different scenarios, including edge cases, to verify the accuracy and reliability of redundancy detection.

- Documentation: The system should include detailed documentation covering all aspects of functionality and setup.

- Performance: The system should perform the redundancy analysis within a reasonable time frame, even with large project backlogs. It should be optimised to process large amounts of data without significant performance degradation.

- Scalability: The system should be scalable to handle an increasing the number of USs and larger project backlogs.

4.3 Design

This section describes the workflow and architectural considerations underlying the framework.

Design Overview

To address the redundancy detection requirements specified in Section 4.2, our system initiates with backlogs annotated with Doccano tool[10]generated by Mosser et al.[11] as the primary input[6]. These annotated USs are used to generate graph transformation rules for each US in the backlog using Henshin API.

Subsequently, these rules serve as inputs for the Henshin Conflict and Dependency Analysis (CDA) tool [37], which automates the identification of potentially redundant US pairs. The output from the CDA tool is then utilized to create a report that compiles information on these potentially redundant pairs. This report, in turn, becomes the input for the evaluation process, which outputs statistical data concerning redundancy among USs.

Figure 12 illustrates how each step in this sequence is interconnected, with the output of one step feeding directly into the next. This diagram effectively demonstrates the toolchain and process workflow, highlighting how each tool transforms artefacts and contributes to the overall objective of redundancy detection.

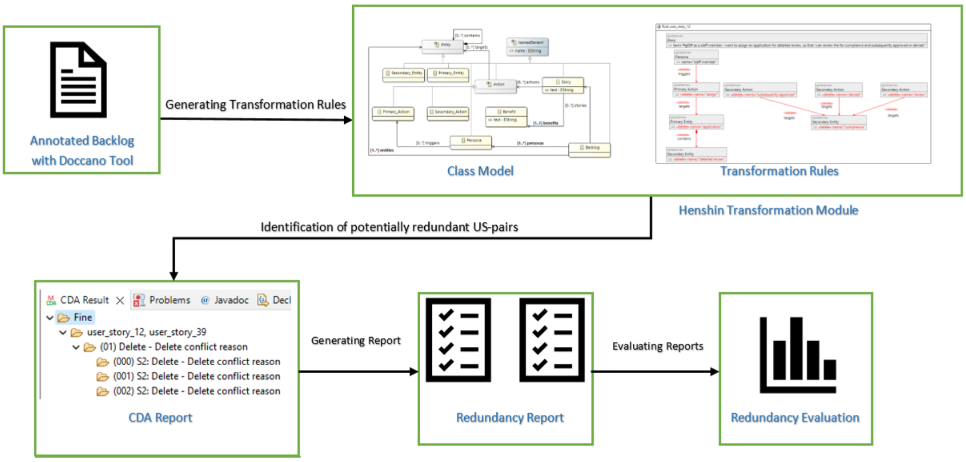


Figure 12: Step-by-step visualisation of the toolchain and its inputs and outputs

US labelling in JSON files As the annotated USs in the JSON files do not contain identifiers, a customised Python script, *nummerise_us.py*, is used to assign an unique identifier to each US, which is stored in a JSON object named *"US_Nr"* [12]. This addition improves the system's ability to distinguish and process individual USs within the analytical pipeline.

Creation of an Ecore meta model The creation of an Ecore meta-model is required to generate rules in Henshin. The constructed meta-model, is based on the conceptual

[10]https://doccano.github.io/doccano
[11]https://github.com/ace-design/nlp-stories
[12]https://github.com/amirrabieyannejad/Redundancy_Analysis/blob/main/Skript/nummerize

meta-model outlined in Figure 5 introduced by Mosser et al. and ensures consistency with the JSON-structured data. This correspondence between the meta-model and the JSON representation underpins the rule generation and transformation processes of the system.

Rule creation process The transition from identified USs to implementable transformation rules within Henshin is facilitated by the package *org.eclipse. emf.henshin.model. compact*. The class *RuleCreator* within the package *org.henshin.backlog.code.rule* is important in this process as it uses different classes to instantiate transformation rules, nodes, edges and attributes. These components are annotated with actions such as *Delete*, *Preserve*, which are necessary for the subsequent application of CDA tool to recognise redundant US-pairs. This methodological approach improves the system's ability to detect redundancies and thus optimises the efficiency of the residue analysis process.

Conflict and Dependency Analysis (CDA) After the creation of transformation rules and the creation of the corresponding Henshin file by the *RuleCreator* class, the system is able to perform a CDA to identify potential redundancy pairs.

Extraction of text reports The creation of a text report is a crucial step in consolidating the results of the CDA, which aims to highlight essential information such as the identification of potentially redundant US-pairs, the enumeration of redundancy clauses, the categorisation of these clauses within the main or benefit parts of the USs and a tabulation of potentially redundant pairs alongside the total number of redundancy clauses.

Evaluating the reports After creating the reports for datasets, we are now able to evaluate the level of redundancy between US-pairs based on JSON reports.

The evaluation process involves a detailed examination of relations such as triggers, targets, and contains within the USs, comparing them to identify exact matches or significant overlaps that could indicate redundancy.

Software Architecture

In this section, we present the basic structures of our workflow and the discipline of creating such structures. Each structure comprises software elements, relations among them, and properties of both.

- Annotated USs with Doccano Tool[13]: Mosser et al. used publicly available requirements from Dalpiaz et al.[13] consisting of 22 product backlogs and 1,679 USs. The dataset is a raw archive of 22 text files, each containing one US per line. As there were no public expert-based annotations, Mosser et al. annotated the

[13] https://github.com/doccano/doccano

dataset using the Doccano tool for *Named Entity Recognition* (NER). Based on their metamodel of the domain, they have labelled the elements of USs as persona, action, entity, benefit part and relationships such as triggers, targets and contains [40].

- Eclipse as IDE[14]: Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. We chose this IDE because it offers the Henshin tool specifically for model-based development.
- Eclipse Modelling Project[15]: It focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modelling frameworks, tooling, and standards implementations.
- Eclipse Modelling Framework (EMF)[16]: The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.
- Henshin[17]: Henshin is an in-place model transformation language for the EMF. It supports direct transformations of EMF model instances (endogenous transformations), as well as generating instances of a target language from given instances of a source language (exogenous transformations).

  Because Henshin enables the specification of restrictions and conditions within graph transformation rules, we use it to enforce and verify US requirements to ensure that the restrictions are met.
- Henshin API: It provides the specification and execution of transformation modules, units and rules. This API is beneficial due to the dynamic creation and modification of transformation modules as part of an automated tool chain[18].
- RuleCreator Class: This class developed within the *org.henshin.backlog.code.rule* package, serves as an integral component of our software architecture, leveraging the Henshin API to facilitates the programmable generation of graph transformation rules. It operates by reading JSON files, which specify USs and their associated actions, entities, and relationships, then systematically constructs corresponding Henshin module, rules, nodes, and their attributes.

  the RuleCreator class effectively bridges the gap between high-level requirements specified in JSON and the low-level execution capabilities of the Henshin, ensuring

[14] https://eclipseide.org
[15] https://eclipse.dev/modeling/
[16] https://eclipse.dev/modeling/emf/
[17] https://wiki.eclipse.org/Henshin
[18] https://wiki.eclipse.org/Henshin/Compact_API

that USs translated into executable transformation rules that reflect the specified software behaviour.

- Henshin's CDA Feature[19]: Henshin's CDA feature can be used to analyse conflicting overlaps of rule applications, which we will interpret as redundancies.

  After the rules and the corresponding Henshin file have been created by the RuleCreator class as artefact, we are now able to pass them to the CDA to find potential redundancy pairs.

  Since the analysis related to the *Attribute* is not yet included in the CDA API, we decided to use the user interface (UI) of the CDA extension by interactively using CDA.

- ReportExtractor Class: The class from the *org.henshin.backlog.code.report* package is used in our software architecture to extract and format reports from the CDA-generated *Minimal-Model.ECore* that contain all information about redundant US-pairs such as redundant elements with their name and type. It uses classes from the *org.eclipse.emf.ecore* package to handle EMF resources that support the management and manipulation of *Minimal-Model.ECore* data in a structured format.

  In operation, the ReportExtractor class reads minimal-model.ecore files containing detailed information about redundant US-pairs. It then processes this data to generate reports in both text and JSON formats, aiding in the systematic analysis of potential redundancies within the USs. This is facilitated through methods that dynamically read and interpret the JSON data, extracting key information such as actions, entities, and their interactions.

- Evaluation Class: The class is part of the *org.henshin.backlog.code.evaluation* package, serves a critical function within our software architecture, focusing on the analysis and determination of redundancies between USs based on specified criteria. This class utilizes JSON processing to assess the overlap and redundancy between elements of USs, such as triggers, targets, and contains, which are vital for identifying potential redundancies in the USs.

  Key functionalities of this class include reading and interpreting reports, where it extracts detailed elements related to USs and evaluates them against redundancy criteria defined in its methods.

Figure 13 shows the architectural composition, highlighting the integral components and their user interface and artefacts.

Regarding redundancy, some definitions are clarified:

Definition 4.1 (**Main and Benefit Parts in User Story**). *A user story (US) is divided into two distinct parts that collectively describe what the user wants, why they want it, and how it will benefit them. These are:*

[19]https://wiki.eclipse.org/Henshin/Conflict_and_Dependency_Analysis

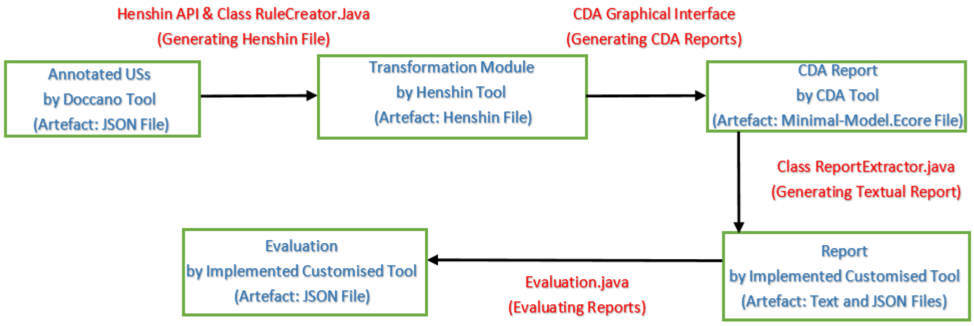


Figure 13: Design phases

- *The main part, which is essential as it clearly and concisely summarises the persona, the intended functionality and the resources required to perform the action. This part usually has follow the format: "As a [persona], I want [what]".*

  Here, the persona helps contextualise the requirement by linking it to a user type, which promotes understanding. The intended functionality describes the action that the persona wants to perform or the function they need, providing a clear statement of the requirement.

  Specifying the resources required to perform the action helps with planning and resource allocation and ensures that the development team is aware of the tools, technologies and time required.

- *The benefit part, which formulates the potential benefit for the end user and typically begins with the phrase "so that". This part of the US is used for justifying the need for a feature by explaining the value or improvement it brings to the user's experience. It connects the functionality directly to user satisfaction, efficiency, or productivity gains, making it easier for the development team to prioritize features based on their impact.*

  It is worth noting that sometimes the benefit part may not exist in the structure of USs. In such cases, the main part can stand alone.

Definition 4.2 (**Clause**). *In this context, a clause is constructed as a pair of words, each pair being linked by a set of predefined relational constructs. These constructs, which are central to the contextual structuring of USs, are described below:*

- *Triggers: A relationship between a persona and an action that includes the invocation of the action by the persona within the narrative of a US.*
- *Targets: A relationship between an action and an entity means that the action influences the entity in question.*
- *Contains: A relation between two entities, indicating that one entity contains another one, in which one entity merges into or is enclosed by another.*

Including the specified clauses in the analysis framework ensures that the words identified in potentially redundant US-pairs are linked by one of the relational constructs outlined: Triggers, Targets, or Contains.

This methodological approach supports the identification of redundancy not only on the basis of lexical similarity, but through the contextual relationships that determine the interactions within the US narratives.

Example 4.1. *When evaluating the textual similarities between two USs, consider the following pair:*
user_story_01: "As a Staff member, I want to manage ordinances, so that I can maintain accurate ordinance information in the System."
user_story_02: "As a Staff member, I want to manage affidavits, so that I can ensure compliance with the requirements prior to the hearing."

Upon comparing these two USs, we identify shared phrases: "Staff member" and "manage." These common elements indicate that both user stories involve the same persona ("Staff member") engaged in a similar type of action ("manage"). However, the focus of their management tasks differs significantly, with one managing "ordinances" and the other "affidavits."

This distinction in the resources being managed suggests that while there are textual overlaps and commonalities in persona and action, the USs are not duplicates, as they address different aspects of the staff member's responsibilities and goals within the organization.

Definition 4.3 (Redundancy). *Redundancy refers to situations where all phrases of USs duplicate or largely overlap with others. This is the case when only a part (main or benefit) or entire USs are syntactically identical, meaning that some or all phrases and their orderings match perfectly between two USs.*

Notation. *Lowercase identifiers refer to single elements, and uppercase identifiers denote sets. A user story us is a 2-tuple $us = \langle m, b \rangle$ where:*

- *A main m is a 6-tuple $m = \langle p, A, E, Tr, Ta, Co \rangle$ which p is the persona, $A = \{a_1, a_2, ...\}$ is a set of actions, $E = \{e_1, e_2, ...\}$ is a set of entities, triggers references $Tr = \{(p_1, a_1), (p_2, a_2), ...\}$ is a set of pairs of persona and action, targets references $Ta = \{(a_1, e_1), (a_2, e_2), ...\}$ is a set of pairs of action and entity, and contains references $Co = \{(e_1, e_2), (e_*, e_*), ...\}$ is a set of pairs of entities.*
- *A benefit b is a 4-tuple $b = \langle A, E, Ta, Co \rangle$ where $A = \{a_1, a_2, ...\}$ is a set of actions, $E = \{e_1, e_2, ...\}$ is a set of entities, $Ta = \{(a_1, e_1), (a_2, e_2), ...\}$ is a set of pairs of action and entity, $Co = \{(e_1, e_2), (e_*, e_*), ...\}$ is a set of pairs of entities.*

To denote that a syntactic operator, we add the subscript "syn"; for instance, $=_{syn}$ is syntactic equivalence which introduced by Lucassen et al. [35].

In the following, let $us_1 = \langle m_1, b_1 \rangle$ *which* $m_1 = \langle p_1, a_1, e_1, tr_1, ta_1, co_1 \rangle$ *and* $b_1 = \langle a_1, e_1, ta_1, co_1 \rangle$ *as well as* $us_2 = \langle m_2, b_2 \rangle$ *which* $m_2 = \langle p_2, a_2, e_2, tr_2, ta_2, co_2 \rangle$ *and* $b_2 = \langle a_2, e_2, ta_2, co_2 \rangle$.

The entity e_1 *is an exact redundant of entity* e_2*, when the entities are identical. Formally,*
$isRedundant(e_1, e_2) \leftrightarrow e_1 =_{syn} e_2$

The action a_1 *is an exact redundant of action* a_2*, when the actions are identical. Formally,*
$isRedundant(a_1, a_2) \leftrightarrow a_1 =_{syn} a_2$

The triggers reference tr_1 *is an exact redundant of trigger* tr_2*, if they run between redundant personas* p_1 *and* p_2 *and redundant actions* a_1 *and* a_2*. Formally,*
$isRedundant((p_1, a_1), (p_2, a_2)) \leftrightarrow tr_1 =_{syn} tr_2$

The targets reference ta_1 *is an exact redundant of target* ta_2*, if they run between redundant actions* a_1 *and* a_2 *and redundant entities* e_1 *and* e_2*. Formally,*
$isRedundant((a_1, e_1), (a_2, e_2)) \leftrightarrow ta_1 =_{syn} ta_2$

The contains reference co_1 *is an exact redundant of contain* co_2*, if they start at redundant entities* e_1 *and* e_2 *and end in redundant entities* e_3 *and* e_4*. Formally,*
$isRedundant((e_1, e_3), (e_2, e_4)) \leftrightarrow co_1 =_{syn} co_2$

To comprehensively assess redundancy, it is important to consider not only the textual content, but also the functional and contextual relevance of each phrase within the USs. By analysing triggers, targets and contains relationships, we can uncover redundancies that may not be immediately apparent through a simple text comparison.

Definition 4.4 (**Full Redundancy**). *A US-pair are fully redundant in their main parts if*

- *there are redundant targets references* Ta_2 *in* m_2 *for all targets references* Ta_1 *in* m_1*, so that* Ta_1 *and* Ta_2 *are redundant, and vice versa, and*
- *there are redundant triggers references* Tr_2 *in* m_2 *for all triggers references* Tr_1 *in* m_1*, so that* Tr_1 *and* Tr_2 *are redundant, and vice versa, and*
- *there are redundant contains references* Co_2 *in* m_2 *for all contains references* Co_1 *in* m_1*, so that* Co_1 *and* Co_2 *are redundant, and vice versa.*

A US-pair are fully redundant in their benefit parts if

- *there are redundant targets references* Ta_2 *in* b_2 *for all target references* Ta_1 *in* b_1*, so that* Ta_1 *and* Ta_2 *are redundant, and vice versa, and*
- *there are redundant contains references* Co_2 *in* b_2 *for all contain references* Co_1 *in* b_1*, so that* Co_1 *and* Co_2 *are redundant, and vice versa.*

A US-pair was categorised as "full redundancy" in the main/benefit part if all occurring clauses in the main/benefit part consisting of triggers (for the main part only), targets and contains were syntactically identical.

This means that in each pair, the wording, order and structure of the clauses relating to the triggers and targets must be identical to fall into this category. The check also extends to the "contain" elements, if these are present, to ensure that these also match perfectly and without deviations.

Full redundancy in the main part means that the redundant stories do not provide additional information and can be consolidated or eliminated without compromising the completeness or operational integrity of the system specifications.

On the other hand, having this level of redundancy in the benefit parts means that USs achieve the same goal; they can be categorised in the same group. This means that if the end results or benefits described by the USs are identical, regardless of the different actions/entities or triggers that lead to these results, the USs effectively serve the same purpose within the system.

Example 4.2. *For example, the following US-pair is identified as "Full Redundancy" in the main part:*

user_story_01: "#g14# as a #publisher#, i want to #publish# a #dataset#, so that I can view just the dataset with a few people."

user_story_02: "#g14# as a #publisher#, i want to #publish# a #dataset#, so that I can share the dataset publicly with everyone."

All existing clauses associated with the main elements in this scenario - in particular the triggers (where "Publisher" is the persona and "Publish" is the action) and the targets (where "publish" is the action and "Dataset" is the entity) - are exactly the same in the main parts of the USs. This complete similarity shows that there is "full redundancy" in the main parts of these USs.

Example 4.3. *Following US-pair is identified as "Full Redundancy" in "Benefit" part:*

user_story_02: #g05# as a data publishing user, I want to be able to edit the model of data I have already imported, so that I can #fix# #bugs# or #make# #enhancements# in the #API# built for my #data#.

user_story_07: #g05# as a data publishing user, I want to be able to edit the data source of data I have already imported, so that I can #fix# #bugs# or #make# #enhancements# in the #API# built for my #data#.

As we can see, targets(["fix", "bugs"], ["make", enhancements"]) and contains (["API", "enhancements"], ["data","API"]) in the benefit part are identical between USs, therefore, we have "Full Redundancy" in the benefit parts.

Last but not least, sometime there are words in the annotated USs that are not labelled as a reference (triggers, targets, or contains) by the Doccano tool at all, which leads to a full redundancy being incorrectly recognised instead of a partial.

Example 4.4. *Considering following US-pair:*

user_story_26: #g05# as an #api user#, I want to be able to #understand# if a #user# is an administrator, so that [...].

user_story_25: #g05# as an #api user#, I want to be able to #understand# if a #user# is a publisher, so that [...].

In this example, full redundancy is found due to the redundant targets and triggers in the main part, which is not entirely correct. This is because there are some phrases that do not labelled as contains or targets in any relations, especially: "administrator" (user_story_26) and "publisher" (user_story_25).

Definition 4.5 (**Partial Redundancy**). *A US-pair are partially redundant in their main parts if there is a target reference ta_1 in m_1 and a target reference ta_2 in m_2, so that ta_1 and ta_2 are redundant.*

A US-pair are partially redundant in their benefit parts if there is a target reference ta_1 in b_1 and a target reference ta_2 in b_2, so that ta_1 and ta_2 are redundant.

Note that in this definition, a US-pair that is fully redundant in the main part (or in the benefit parts) is also partially redundant in the main part (or in the benefit parts).

Partial redundancy in USs occurs when only certain clauses, such as targets, have significant overlap but are not completely identical. This means that while there are shared aspects such as targets between USs, other clauses such as triggers or contains may not overlap, indicating an incomplete match. Such a scenario indicates that there is substantial, but not fully a match between the USs.

Example 4.5. *Following US-pair is identified as "Partial Redundancy" in the main parts:*

user_story_09: #g04# as a #user#, I want to be able to #view# a #map display# of the public recycling bins around my #area#.

user_story_10: #g04# as a #user#, I want to be able to #view# a #map display# of the special waste drop off sites around my #area#.

As we can see, there are some redundancy clauses such as triggers (["user", "view"]), targets (["view", "map display"]) and contains (["map display", "area"]) between the USs. There are also clauses such as contains (["public recycling bins"] vs. ["hazardous waste collection points"]) that are distinct elements justifying the maintaining of separate USs. Therefore, we assess this as "Partial redundancy" in the "Main parts".

Example 4.6. *Following US-pair is identified as "Partial Redundancy" in benefit parts:*

user_story_17: #g03# as a staff member, I want to manage approved proffers, so that I can #ensure# #compliance# with and satisfaction of the proffer in the future.

user_story_30: #g03# as a staff member, I want to manage affidavits, so that I can #ensure# #compliance# with the requirements prior to the hearing.

As we can see, there is a redundancy clause as targets ([”ensure”, ”compliance”]) between the USs in the benefit parts, but there is also a clause as contains ([”proffer”, ”satisfaction”] vs [”requirements”, ”hearing”]), which leads us to evaluate this as “Partial Redundancy” in the “Benefit” part.

Design Phases

To provide a comprehensive overview of the design phases, this section explains each step of the process, from initial setup to final evaluation, using practical examples.

Step 1: Data Preparation

As primary input, we receive a graph-based model generated by the Doccano tool, which represents the refined and annotated dataset for the recognition of *entities*, *actions*, *personas*, and *benefits* of USs [6].

The datasets have the JSON format, the structure of which is very important in the Java classes, namely *RuleCreator*, *ReportExtractor*, and *Evaluation*. Therefore, understanding the JSON format provided is needed for the further procedure.

Each JSON file for a backlog dataset contains a JSON-array in which each US entry is defined as a JSON-object. Listing 1 illustrates the format used for the US entry.

```
[{
"Text": "...",
"Persona": [ "..." ],
"Action": {
        "Primary Action": [ "...", ... ],
        "Secondary Action": ["..."], ... },
"Entity": {
        "Primary Entity": [ "...", ... ],
        "Secondary Entity": [ "...", ...] },
"Benefit": "...",
"Triggers": [ [ "...", "..." ] ],
"Targets": [ [ "...", "..." ], ... ],
"Contains": [ [ "...", "..." ], ... ]
},...]
```

Listing 1: The JSON format of each US entry in JSON file

Mosser et al. have linked each *Persona* to each *Primary Action* as *Trigger* relationships, each *Primary Actions* to each *Primary Entity* as *Target* relationships and each *Primary/Secondary Entity* to each *Primary/Secondary Entity* implying a *Contains* relationship[6]. To interact with the entries in JSON file, we need to distinguish between the entries that are defined as JSON-objects, such as: Text, Action, Entity, Benefit and the entries that are defined as a JSON-array, such as: Persona, Primary/Secondary Action, Primary/Secondary Entity, Triggers, Targets, and Contains.

Identifying USs in JSON-File

Annotated USs in each JSON file have no identifier. To distinguish USs, we use a Python script called *nummerise_us.py* [20], which receives JSON files as input and adds a JSON object named "US_Nr" with an identifier as value (e.g. user_story_01) to each US and returns the JSON files as output. Listing 2 illustrates the added JSON object "US_Nr" and its value in the JSON file.

```
[{
"Text": "...",
"Persona": [ "..." ],
"Action": {
        "Primary Action": [ "...", ... ],
        "Secondary Action": ["..."], ... },
"Entity": {
        "Primary Entity": [ "...", ... ],
        "Secondary Entity": [ "...", ...] },
"Benefit": "...",
"Triggers": [ [ "...", "..." ] ],
"Targets": [ [ "...", "..." ], ... ],
"Contains": [ [ "...", "..." ], ... ],
"US_Nr" : "user_story_01"
},...]
```

Listing 2: The JSON format with the additional JSON object "US_Nr" and its value

Step 2: Creation of Rules

Step 2 of the design involves a central process in which the US data structured in JSON files is transformed into transformation rules using the Henshin API. This involves the creation of an Ecore meta-model that represents the structure of the data we are working with, followed by the generation of Henshin transformation rules using RuleCreator class.

Creating Ecore Meta-Model

To be able to create rules in Henshin, an Ecore (meta)-model should be available. Ecore is the core (meta)-model at the heart of the EMF. It enables the formulation of other models by utilising its constructs.

Accordingly, we create an Ecore meta-model as shown in Figure 14, which is inspired by the meta-model shown in Figure 5 represented by Mosser et al. and corresponds to the JSON-objects in the JSON-file as follows:

- *Persona* as a class in the meta-model corresponds to the JSON-object "Persona" in the JSON-file.

[20] https://github.com/amirrabieyannejad/Redundancy_Analysis/tree/main/Skript/nummerize_us

- *Entity* as an abstract class, from which *Primary/Secondary Entity* inherits as a class in the meta-model, corresponds to the JSON-object “Entity”, which contains two JSON-arrays, namely “Secondary/Primary Entity” in the JSON-file.
- *Action* as an abstract class and *Primary/Secondary Action* as an inherited class in the meta-model correspond to the JSON-object “Action”, which contains two JSON-arrays, namely “Secondary/Primary Action” in the JSON-file.
- *Benefit* as a class in the meta-model, which also has an attribute called “text” that corresponds to the JSON-object “Benefit” in the JSON-file.
- *Story* as a class in the meta-model that contains text from US, which also has an attribute called “text” that corresponds to the JSON-object “Text” in the JSON-file.
- Abstract class *NamedElement* has attribute *name*, which Primary/Secondary Action/Entity inherit from it, which corresponds to the value of *Primary/Secondary Action/Entity* in JSON-file.
- *Edge* with the name *triggers* between Persona and Primary Action in the meta-model, which corresponds to the JSON-array “Triggers”, where each JSON-array in it contains a pair, the first element corresponding to the *Persona* and the second to the *Primary Action*.
- *Edge* named *targets* between Primary/Secondary Action and Primary/Secondary Entity in the meta-model, which corresponds to the JSON-array “Targets”, where each JSON-array has a pair, the first element corresponding to “Primary/Secondary Action” and the second element corresponding to “Primary/Secondary Entity”.
- *Edge* named *contains* between Primary/Secondary Entity and itself in the meta-model, which corresponds to the JSON-array “Contains”, where each JSON-array in it has a pair where the first element corresponds to “Primary/Secondary Entity” and the second element corresponds to “Primary/Secondary Entity”.

Creating Rules

With the identified USs in the the JSON-file, we generate rules with the Henshin package *org.eclipse.emf.henshin.model.compact*, which is responsible for the creation of *transformation rules* and their *classes*, *attributes*, and *edges* and annotates them with <*Delete*>, or<*Preserve*>, which are vital for the CDA tool to recognise the redundant pairs.

To generating rules we create a package named *org.henshin.backlog.code.rule* and specially the class *RuleCreator* which used following classes[21]:

[21] https://wiki.eclipse.org/Henshin/Compact_API

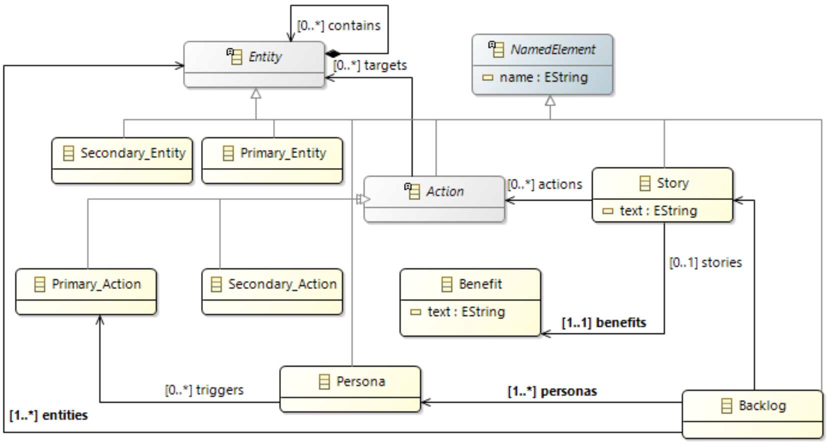


Figure 14: Ecore meta-model inspired by Mosser et al. [40]

- *org.eclipse.emf.henshin.model.compact.CModule*: CModule class can import elements from an Ecore file to use them in the transformation process responsible for linking the Ecore meta-model to the Henshin-file to be created.

- *org.eclipse.emf.henshin.model.compact.CRule*: Once we have a CModule, we can specify transformation rules with the CRule class and create them.

- *org.eclipse.emf.henshin.model.compact.CNode*: Now that we have a transformation rule, we want to fill this rule with nodes, edges and attributes. To create a node within a transformation rule, we need the CRule class. To create an edge we need to reference two nodes together. The default action when specifying a node, edge or an attributes is the <*preserve*>action. We can also specify a different action when we create a node or an edge, specially <*delete*>.

- org.henshin.backlog.code.rule.RuleCreator: We implement *RuleCreator* class that creates a rule with annotated nodes, edges and attributes based on a JSON-file as input and a Henshin-file containing all rules as output, where each rule and its members (nodes, attributes, edges) correspond to the individual US and their JSON-objects/arrays in the JSON-file.

  The most important design decision of this class is the way nodes, attributes and edges are annotated in order to be able to apply CDA in rules stored as a Henshin file.

  We decided to annotate the “name” attribute of all Primary/Secondary Actions/Entities and their associated edges including “targets”, “triggers” and “contains” as `<delete>`action. The main goal is to increase the probability of identifying US-pairs characterised by matching names of *action* nodes and *entity* nodes in conjunction with an edge called *targets*. This congruence serves as a basic criterion for identifying potentially redundant US-pairs and simplifies the process of redundancy detection in the context of US analysis.

Example 4.7. *Listing 3 shows the JSON format in relation to user_story_12 and Figure 15 shows the application of the RuleCreator class in this US, which is a transformation rule where the targets and the associated contains relationships are annotated as a <delete>action and the rest of the nodes and edges are annotated as a <preserve>action.*

Text of US is: user_story_12: "#G03# As a Staff member, I want to assign an application for detailed review, so that I can review the for compliance and subsequently approved or denied."


```
{
    "Text": "#G03# As a Staff member, I want to Assign an Application
    for Detailed Review, so that I can review the for compliance and
    subsequently approved or denied.",
    "Persona": ["Staff member"],
    "Action": {"Primary Action": ["Assign"],
    "Secondary Action": ["subsequently approved","denied","review"]},
    "Entity": {"Primary Entity": ["Application"],
    "Secondary Entity": ["compliance","Detailed Review"]},
    "Benefit": "I can review the for compliance and subsequently
    approved or denied",
    "Triggers": [["Staff member","Assign"]],
    "Targets": [["Assign","Application"],["review","compliance"],
    ["subsequently approved","compliance"],["denied","compliance"]],
    "Contains": [["Application","Detailed Review"]],
    "US_Nr": "user_story_12"
  },
```


Listing 3: JSON object related to user_story_12

As we can see, the "targets" edges and their direct relationships ("triggers" and "contain", if any) are also annotated as <delete>, which is very important to find redundant elements with the CDA tool.

Example 4.8. *Listing 4 shows the JSON object related to user_story_39 and Figure 16 shows transformation rule generated by RuleCreator class.*

Text of US is: user_story_39: "#G03# As a plan Review Staff member, I want to review plans, so that I can review them for compliance and either approve, or fail or deny the plans and record any conditions, clearances, or corrections needed from the applicant."

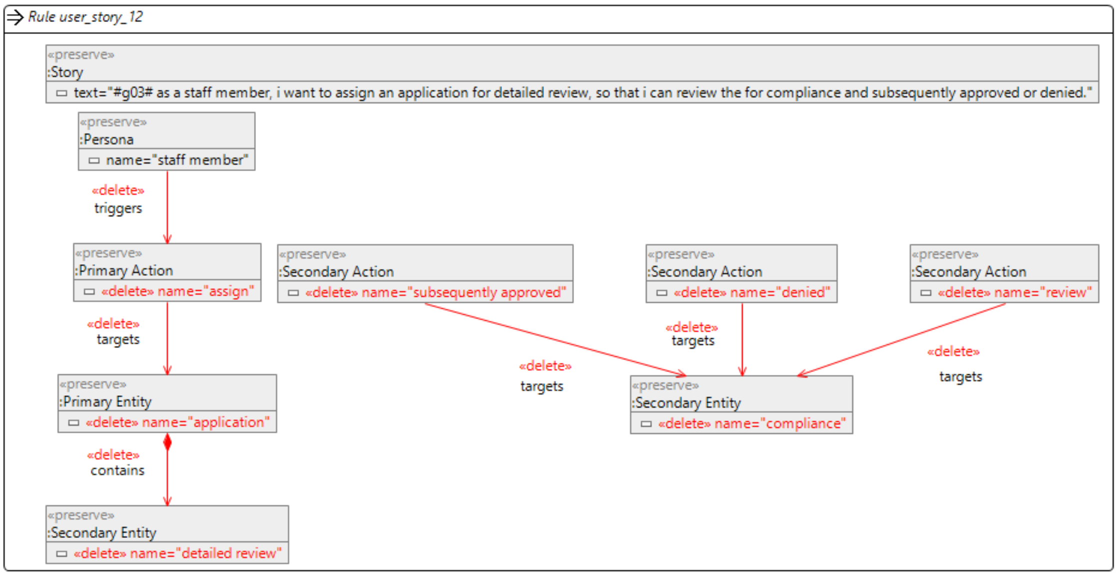


Figure 15: Generated transformation rule related to user_story_12 using RuleCreator class

```
{
 "Text": "#G03# As a Plan Review Staff member, I want to Review
 Plans, so that I can review them for compliance and either approve,
 or fail or deny the plans and record any conditions, clearances, or
 corrections needed from the Applicant.",
 "Persona": ["Plan Review Staff member"],
 "Action": {"Primary Action": ["Review"],
 "Secondary Action": ["deny","review","approve","fail","record"]},
 "Entity": {"Primary Entity": ["Plans"],
   "Secondary Entity":["compliance","plans","clearances","Applicant",
   "corrections","any conditions"]},
 "Benefit": "so that I can review them for compliance and either
 approve, or fail or deny the plans and record any conditions,
 clearances, or corrections needed from the Applicant",
 "Triggers": [["Plan Review Staff member","Review"]],    "Targets":[[
 "Review","Plans"],["review","compliance"],["approve","plans"],["fail"
 ,"plans"],["deny","plans"],["record","clearances"],["record",
 "corrections"],["record","any conditions"]],
 "Contains": [["Applicant","plans"],
   ["plans","clearances"],["plans","corrections"],["plans","any
   conditions"]
 ],
 "US_Nr": "user_story_39"
 },
```

Listing 4: JSON object related to user_story_39

Attribute "plan", as we can see, it appears both in the main part as a primary entity and in the benefit part as a secondary entity, forming the various relationships as targets and contains.

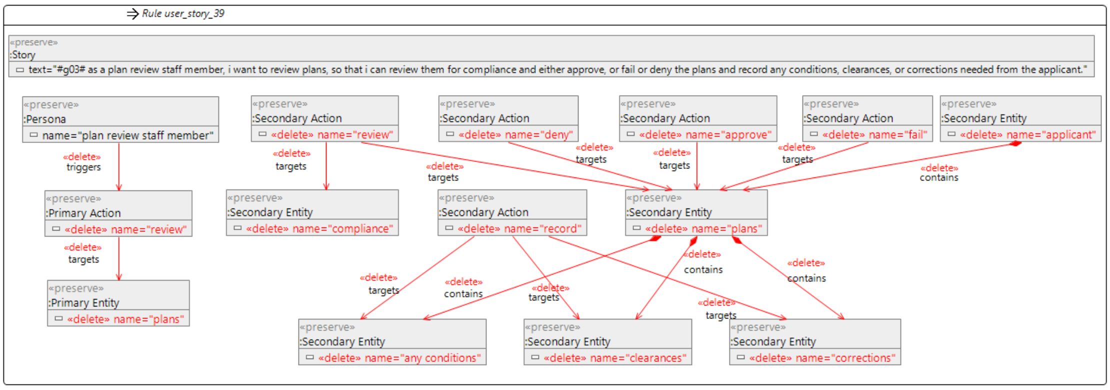


Figure 16: Generated transformation rule related to user_story_39 using RuleCreator class

Example 4.9. *Listing 5 shows the JSON object related to user_story_51 and Figure 17 shows transformation rule generated by RuleCreator class.*

Text of US is: user_story_51: "#G03# As an Enforcement Staff member, I want to issue a notice of violation, so that I can provide formal communication to the responsible party."

```
{
  "Text": "#G03# As an Enforcement Staff member, I want to Issue a
  Notice of Violation, so that I can provide formal communication to
  the responsible party.",
  "Persona": ["Enforcement Staff member"],
  "Action": {"Primary Action": ["Issue"],
    "Secondary Action": ["provide"]},
  "Entity": {"Primary Entity": ["Notice of Violation"],"Secondary
  Entity": ["formal communication","responsible party"]},
  "Benefit": "I can provide formal communication to the responsible
  party",
  "Triggers": [["Enforcement Staff member","Issue"]],"Targets": [[
  "Issue","Notice of Violation"],["provide","formal communication"]],
  "Contains": [],
  "US_Nr": "user_story_51"
}
```

Listing 5: JSON object related to user_story_51

Last but not least, we have determined that the secondary entity "responsible party" has neither a target nor a contains relationship. Therefore, it is not annotated as <delete>, but as <preserve>. This is due to the fact that some phrases have been identified as entities in the Doccano tool, but their relationship is not annotated at all, which is problematic for analysing redundancy.

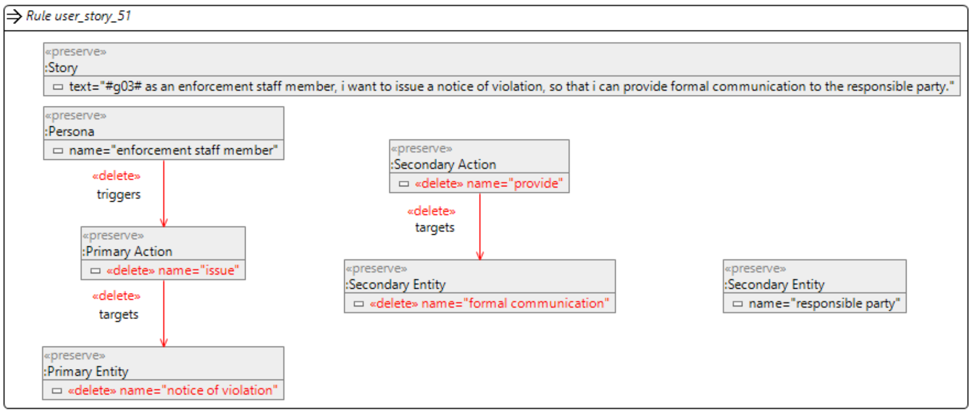


Figure 17: Generated transformation rule related to user_story_51 using RuleCreator class

Step 3: Applying CDA

Now that the rules and the corresponding Henshin file have been created by the RuleCreator class, we can pass them to the CDA to find conflicting rule application overlaps, which we will interpret as potential redundancies.

Since the analysis related to the *attribute* is not yet considered in the CDA API[22], we decided to use the user interface (UI) of the CDA extension of Henshin, which supports analysis of conflict of rules through the interactive use of CDA.

To apply CDA to Henshin files, we just need to right-click on the Henshin file and select *Henshin -> conflict and Dependency Analysis* from the context menu as shown in Figure 18. A user interface then appears, prompting to select the rule sets to be analysed

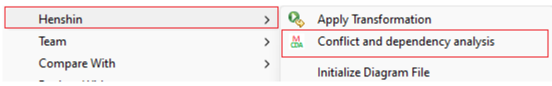


Figure 18: Applying CDA to the selected Henshin file

and the type of analysis. We then select as "*First*" and "*Second Rules*", all rules related to USs. Additionally, as the type of analysis we select "*conflicts*" as illustrated in Figure 19. On the next page of the CDA UI shown in Figure 20, we specify the depth of analysis that we use with "*Fine granularity*" when selecting "*Create a complete result table*" and "*Create an abstract result table*".

Fine granularity provides a detailed examination of each conflicting rule by listing all conflict reasons. Unlike coarse granularity, which focuses only on minimal conflict reasons, fine granularity includes both minimal and more general conflict reasons. The binary granularity, where simple conflicting rule pairs are listed, may be too simple for complex systems where understanding the nature of the conflict is essential for the

[22] https://wiki.eclipse.org/Henshin/conflict_and_Dependency_Analysis

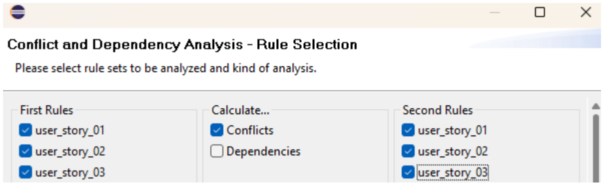


Figure 19: CDA user interface: Selection of rules and type of analysis

solution. We choose "Fine granularity" as the depth of analysis due to the fact that it shows all conflict reasons for each conflicting rule pair. This allows for a deeper understanding of how different model fragments contribute to conflicts.

A conflict reason is a model fragment whose presence leads to a conflict. General conflict reasons result from different combinations of minimal conflict reasons[48]. During

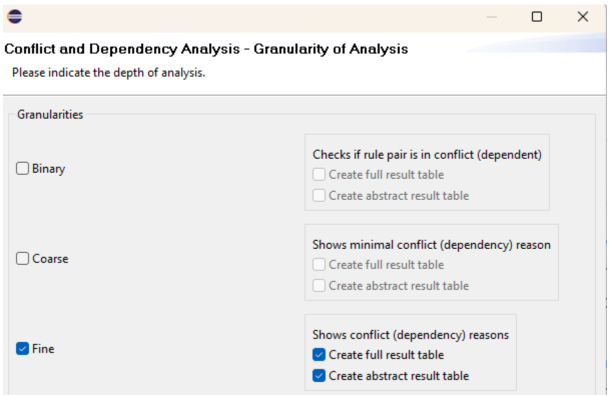


Figure 20: CDA user interface: Selection of report granularity

the execution of the CDA analysis, the rule pairs is analysed and a conflict analysis is performed. Once the calculation is complete, the results are listed in the "CDA" - >"Result window", as shown in Figure 21. The top entry shows the granularity, which in our case is "Fine". These entries contain the rule pairs that conflict with each other. Each rule pair contains a number of conflict reasons. Figure 22 shows how the data is

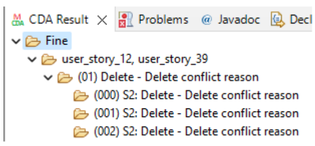


Figure 21: CDA report with fine granularity

saved in the project tree view. The results directory is created in the directory containing the Henshin that was used for the analyses. The new folder name is the date and time at which the analysis was performed. In contrast to the "*CDA/Results*" view, this folder contains all conflict reasons and atoms together in a rule pair directory. For each conflict

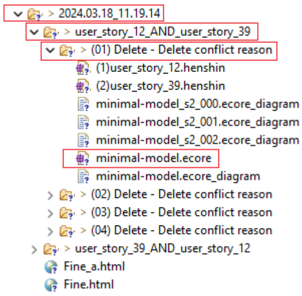


Figure 22: Saving CDA results data in the project structure view

reason, there is a "*minimal-model.ecore*" file, that contains packages in which various conflict elements such as "attributes" and "references" (edges) are mapped together and displayed in different packages.

Figure 23 shows the representation of the redundant attributes and references. An attribute has the property of changing the value and is represented by an arrow "->". The attribute from the first rule is separated from the second rule by an underscore, just as with the nodes.

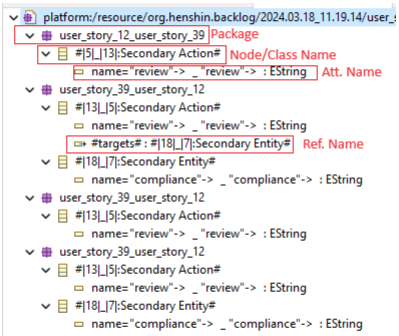


Figure 23: Representation of redundant attributes and references in *minimal-model.ecore* file

Example 4.10. *To illustrate this step, we also pass the transformation rules created in step 2, which reflect three USs (user_story_12/39/51), to the CDA tool and selecting "conflicts" as conflict type to calculate with "fine granularity" as the depth of analysis. Once the calculation is complete, the results are listed in the "CDA" ->"Results Window" as shown in Figure 24. Figure 25 shows how the data is saved in the project's tree view, which contains all conflict reasons and atoms together in a rule pair directory. As we can see, the CDA tool has only found redundancy between user_story_12 and user_story_39. This is because there are no redundant clauses between two USs (user_story_12 and user_story_39) and user_story_51.*
Regarding the redundant elements specifically, we can refer to the created file "minimal-

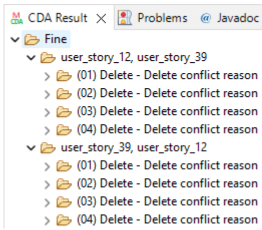


Figure 24: CDA report in relation to three transformation rules with "conflicts" as the type to be calculated and "fine granularity" as the depth of the analysis

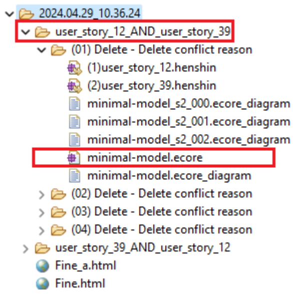


Figure 25: Saving CDA results data in the project's tree view

model.ecore", which is located in the tree view of the project under each conflict reason. Figure 25 show the minimal-model.ecore file related to user_story_12 and user_story_39.

The file Minimal-model.ecore, which refers to user_story_12 and user_story_39, is divided into packages, with each package containing different matches of redundant elements. If there is a redundancy between two elements, this is explicitly indicated by a hash symbol (#). Figure 26 illustrates the redundant elements found in each package.

For example, the attribute "name" with the value "review" in "Secondary Action"

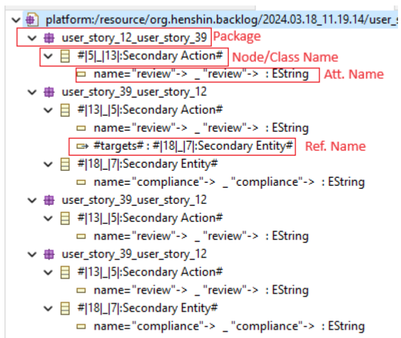


Figure 26: Representation of the redundant elements in each package within *minimal-model.ecore* file

and the attribute "targets" with the value "compliance" in "Secondary Entity" are labelled as redundant (both with a hash symbol). The reference to "targets" is also marked as redundant (with a hash symbol), which means that the reference (targets) from "Secondary Action" to "Secondary Entity" is also redundant between USs, which is a very important criterion for finding valid redundant elements.

Step 4: Report Extraction

To create a lightweight report for the group or individual in question, we need to extract the key information from the CDA report, e.g. redundancy US-pair, redundancy clauses, count of redundancy clauses in each part of the US (main or benefit part), and create a report as a text file with the following information:

- A table of potential redundant pairs with the number of total redundancy clauses.
- Founded potential redundant US-pairs.
- Redundancy words and clauses of founded US-pairs. Clauses consisting of two words that have one of the relationships triggers, targets or contains.
- Text of US-pairs whose redundancy words are marked with a hash symbol (#).
- Parts of the sentence in which words and clauses are found.

Structure of a US

The delineation and checking of redundancy clauses within USs requires a methodical approach, especially when distinguishing between the main and the benefit part of a US-pair. This distinction is crucial to ensure that redundancy identifications within one part are not mistakenly transferred to the other. Consequently, the analytical framework comprises three conditions, each of which specifies its own methodology for case processing:

- Presence of benefit in both USs of the redundancy-pair: if a benefit is identifiable in each US of the pair, a process of separation is used to split the main content from the benefit parts. After this separation, a targeted search for redundancy clauses is carried out only within the main part, whereby identified redundancies are annotated with a hash symbol (#). This process is repeated for the benefit parts to ensure a thorough check and marking of redundancies within each individual part.
- Exclusive presence of a benefit in one US of the redundancy-pair: In scenarios where only one US of the pair contains a benefit part, the analysis is limited to the main part of both USs. The aim remains the identification and annotation of redundancy clauses within this part. The lone benefit part remains in its original state and is excluded from the redundancy check.

- Absence of benefit parts in both USs of the pair: If neither of the two USs of the redundancy-pair contains a benefit part, the focus shifts completely to the main parts. The investigation is designed to highlight redundancy clauses within these parts, whereby the benefit parts are not taken into account due to their non-existence.

This structured and segmented approach ensures precise and efficient identification of redundancy clauses within the USs, optimising the clarity and effectiveness of textual report.

Example 4.11. *After the CDA directory for user_story_12 and user_story_39 has been created by the CDA tool using the UI, we pass the location of the directory to the ReportExtractor class to extract the important information and store it in the text report and the JSON report. Listing 6 illustrates the example of the textual report. In this case, the report only contains one US-pair.*

```
* Table of potential redundancies between user stories and the number of their overlapping
elements

          us_12  us_39          USs identifiers
    us_12   0      1            Total redundant clauses
    us_39   1      0

-----------[Potential Redundant User Stories found]-----------
{user_story_12_AND_user_story_39}
                                                                         USs Texts
Redundant clauses within user stories are:
* Secondary Action: review
* Secondary Entity: compliance                          Redundant Clauses
* Targets: Link from "review" to "compliance" is found.

user_story_39: #g03# as a plan review staff member, i want  to review plans, so that i can
#review# them for #compliance# and either approve, or fail or deny the plans and record any
conditions, clearances, or corrections needed from the applicant.

user_story_12: #g03# as a staff member, i want to assign an application for detailed review,
so that i can #review# the for #compliance# and subsequently approved or denied.

The following sentence parts are candidates for possible redundancies between user stories:
                                                                         Parts of sentences
user_story_12:  so that i can #review# the for #compliance# and subsequently approved or
denied.
user_story_39:  so that i can #review# them for #compliance# and either approve, or fail or
deny the plans and record any conditions, clearances, or corrections needed from the
applicant.
```

Listing 6: Example of generated textual report for one US-pair

As we can see, the text report consists of a 2 x 2 table whose first column and first row are US identifiers, and the numbers inside the table are the total number (benefit part + main part) of redundancy elements between two USs; secondly, the redundant clauses related to the redundant US-pair are listed; thirdly, the text of US whose redundant phrases are marked with a hash symbol; finally, the part of the clauses in which redundant elements occur is displayed.

For further evaluation purposes and easy export of the report to another platform such as Excel, a JSON report is created that collects the information about redundant US-pairs separately in a JSON object with the following entries:

- Potential Redundant User Stories: which has stored the US-pair identifier(e.g. "user_story_12_AND_user_story_39").
- Status: consisting of "Main/Beneift Part Redundancy Clauses" and "Total Redundancy Clauses", which store the count of redundancy clauses in the main and benefit part as well as in the total part of the US.
- Entity: which can consist of a "Secondary/Primary Entity" and stores the founded redundant entities.
- Common Targets/Contains: which consists of the "Main Part" and "Benefit Part" entries and only stores the targets/contains relationships that are common between the USs in a particular part of the USs. For example, if there are common redundant targets in the main part of the USs, these are included in the "Main Part" entity of the "Common Targets".
- Text: consisting of two entries, namely "First UserStory" and "Second UserStory", in which the text of the US-pair whose hash symbol has already been applied in redundant phrases is stored.
- Project Number: stores the the Project identifier (e.g. "G03").
- Part of Sentence: consists of the entries "First UserStory" and "Second UserStory", in which the part of the US sentences containing redundant clauses is stored.
- All Targets/Contains: which consists of the "Main Part" and "Benefit Part" entries and stores the whole targets/contains relationships that are occurred in the particular part of the USs.

Example 4.12. *Listing 7 illustrates the example of the JSON report regarding user_story_12 and user_story_39.*

```
[
    {
        "Potential Redundant User Stories": "user_story_12_AND_user_story_39",

        "Status": {"Main Part Redundancy Clauses": 0,"Total Redundancy Clauses": 1,
        "Benefit Part Redundancy Clause": 1,},

        "Entity": {"Secondary Entity": ["compliance"]},

        "Common Targets": {"Main Part": [],"Benefit Part": [["review","compliance"]]},

        "Action": {"Secondary Action": ["review"]},

        "Common Contains": {"Main Part": [],"Benefit Part": []},"Targets": [["review",
        "compliance"]],

        "Triggers": [],

        "Text": {"Second UserStory": "user_story_39: #g03# as a plan review staff member,
        i want to review plans, so that i can #review# them for #compliance# and either
        approve, or fail or deny the plans and record any conditions, clearances, or
        corrections needed from the applicant.",
            "First UserStory": "user_story_12: #g03# as a staff member, i want to assign
            an application for detailed review, so that i can #review# the for
            #compliance# and subsequently approved or denied."},

        "Project Number": "g03",

        "Part of Sentence": {"Second UserStory": [" so that i can #review# them for
        #compliance# and either approve, or fail or deny the plans and record any
        conditions, clearances, or corrections needed from the applicant."],
            "First UserStory": [" so that i can #review# the for #compliance# and
            subsequently approved or denied."]},

        "All Contains": {"user_story_39": {"Main Part": [],"Benefit Part": [["applicant",
        "plans"],["plans","clearances"],["plans","corrections"],["plans","any conditions"
        ]]},
            "user_story_12": {"Main Part": [["application","detailed review"]],"Benefit
            Part": []}},

        "All Targets": {"user_story_39": {"Main Part": [["review","plans"]],"Benefit Part"
        : [["review","plans"],["review","compliance"],["approve","plans"],["fail","plans"
        ],["deny","plans"],["record","clearances"],["record","corrections"],["record",
        "any conditions"]]},
        "user_story_12": {"Main Part": [["assign","application"]],"Benefit Part": [[
        "review","compliance"],["subsequently approved","compliance"],["denied",
        "compliance"]]}}
    },

]
```

Listing 7: Example of generated JSON report for one US-pair

Step 5: Report Evaluation

The Evaluation class, part of the *org.henshin.backlog.code.evaluation* package, was developed to determine the level of redundancy in USs based on JSON reports. This class provides methods to evaluate whether two USs are either fully or partially redundant, analysing different application components of these USs.

The evaluation process involves a complex logic to determine whether USs are redundant. This includes:

- Checking whether the arrays are empty or contain similar elements.
- Comparing the individual elements in the arrays for both USs to determine if they fully match (full redundancy) or if they have some common elements (partial redundancy).

Example 4.13. *For the two US-pairs of dataset G03, we apply the Evaluation class to determine whether there is redundancy in the main or benefit part, and if so, what type of redundancy is recognised (full or partial).*

As shown in Listing 8, four entries are added to the JSON report, namely "Main Partially Redundant", "Benefit Part Fully Redundant", "Main Part Fully Redundant", "Benefit Partially Redundant" as "Status" which their value is whether true or false.

```
"Status":
{"Main Part Redundancy Clauses": 0,
"Main Part Partially Redundant": false,
"Total Redundancy Clauses": 1,
"Benefit Part Fully Redundant": false,
"Benefit Part Redundancy Clause": 1,
"Main Part Fully Redundant": false,
"Benefit Part Partially Redundant": true
},
```

Listing 8: Example of generated entries in JSON report regarding evaluation of level of redundancy in main or benefit part

In the case of user_story_12 and user_story_39, the entry "Benefit Partially Redundant" was marked as `true`*, which means that US-pair in benefit parts are partially redundant.*

The class performs these checks by iterating through the JSON arrays of Triggers, Targets and Contains and comparing each element with those in the common sections to determine redundancy.

4.4 Implementation

In this section, we explain the objective and scope of the implementation, the functionality and the programming languages used.

The entire implementation is available in the GitHub repository [23].

[23] https://github.com/amirrabieyannejad/Redundancy_Analysis/tree/main

Methodology

This section explains and introduces tools that are required during the development process.

Following approach and tools are necessary in order to develop our workflow:

- Java as programming language[24]: Java is a widely used object-oriented programming language and software platform that is used to implement the Henshin and EMF APIs, which are critical to our approach to utilising them. Therefore, we use Java as our programming language.
- GitHub as version control[25]: GitHub is a developer platform that allows developers to create, store, manage and share their code. It uses Git software, providing the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.

Implementation Phases

This section contains a step-by-step guide to implementation, starting with the set-up and ending with the final evaluation.

Creation of Rules

In this section, we explain the methods we used to convert the US data structured in JSON files into transformation rules using the Henshin API. The process starts with the creation of an Ecore meta model, as explained in Section 4.3, which reflects the structure of the data. The Henshin transformation rules are then generated using the RuleCreator class and its methods.

Methods of the RuleCreator Class

In this section, the methods of the RuleCreator class is described as follows:

- AssignCmodule: This method assign a CModule to a Ecore meta-model. It creates a new CModule object with the provided Henshin-file name, adds imports from the Ecore file, and returns the module.
- processJsonFile: It takes parsed JSON array as input and processes their attributes, such as persona, actions, entities, text and their edges, such as targets, triggers. Corresponding elements are created as output in a the Henshin transformation module (CModule).

[24] https://www.java.com/de/
[25] https://github.com/

- processRule: It takes the "US_Nr" JSON-object as input and creates a new CRule with the name of unique US identifier in the CModule.
- processPersona: It receives as input the persona extracted from the JSON data and the associated CRule to create a new CNode representing the persona within the provided CRule and adds the attribute "name" with persona as value. Finally, the created CNode representing the persona is returned.
- processText: It receives as input US text extracted from JSON data and the associated CRule to create a new CNode representing the text within the provided CRule and adds the attribute "text" with US text as value. Finally, the created CNode representing the US text is returned.
- processActions: The processActions method is responsible for creating CNode objects that represent actions within the CModule. As parameters, it receives the JSON-object of the actions, the CNode-object representing the persona associated with the actions and the unique identifier of the US. Since the edge triggers only refer to the persona and the primary action, a new CNode is created for each primary action that represents the primary action within the provided CRule, an attribute "name" is added and an edge is created from the persona node to the primary action with the label "triggers". For each secondary action, a new CNode is created to represent the action within the provided CRule and an attribute "name" is added to the action node.
- checkEntityIsTarget: It receives the name of the entity and the JSON-array with information about targets edges. The method iterates through the JSON-array targets, which contains arrays that represent targets edges between actions and entities. It compares the targets entity with the specified entity. If there is a match, true is returned to indicate that the entity belongs to a targets relation.
- processEntities: It receives as parameters the JSON-object with information about the entities, the CRule object representing the US to which the entities belong and the JSON-array with information about the targets associated with the entities. The method checks whether primary/secondary entities are present, then creates a CNode for each primary/secondary entity and checks whether the entity is present in the targets array. If this is the case, its attribute "name" is annotated for deletion. This method is used within the *processContainsEdges* method to determine whether an entity involved in a contains relation is also a targets of another entity.
- processContainsEdges: It receives the JSON-object to be processed, the JSON array with information about contains/targets edges and the US identifier as parameters. It first checks whether both entities belong to contains edges. If both entities exist, an edge is created between them in CRule with the label "contains". If one of the entities is a targets of another entity (as specified in the targets array), the edge is annotated as *delete*. If none of the entities is a targets, the edge is annotated as *preserve*.

Figure 27 is a class diagram that illustrates the attributes, operations and relationships related to the RuleCreator class.

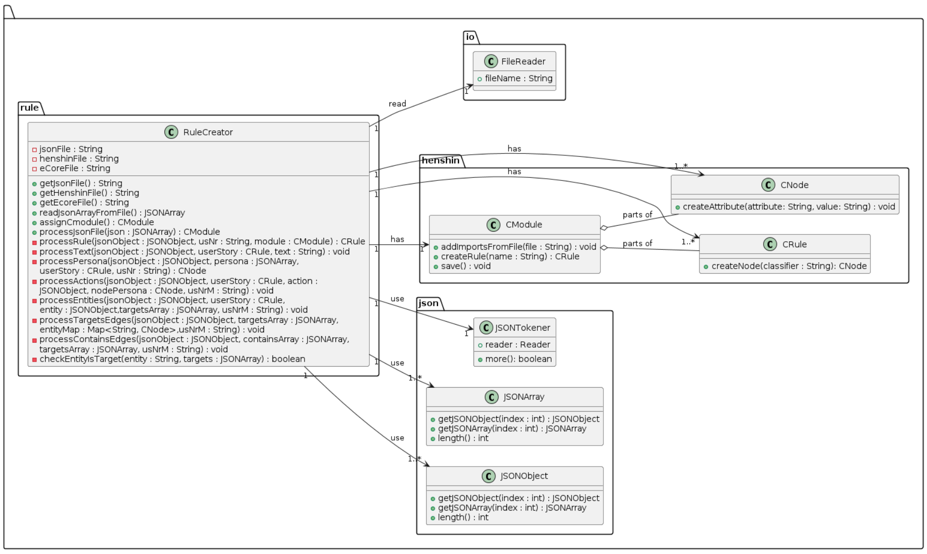


Figure 27: Class diagram of the RuleCreator class and its relationships

Report Extraction

In order to extracting a textual report associated with a specific backlog, we implement a class called *ReportExtractor* within the package *org.henshin.backlog.code.report*, which include the following classes from the package *org.eclipse.emf.ecore*[26]. These classes are important for reading the content of *minimal-model.ecore*:

- org.eclipse.emf.ecore.resource.Resource: A resource of an appropriate type is created by a resource factory; a resource set indirectly creates a resource using such a factory. A resource is typically contained by a resource set, along with related resources.
- org.eclipse.emf.ecore.resource.ResourceSet: A resource set manages a collection of related resources and notifies about changes to this collection. It provides a tree of content. A collection of adapter factories supports the search for an adapter via a registered adapter factory.
- org.eclipse.emf.ecore.EObject: EObject is the root of all modelled objects, therefore all method names start with "E" to distinguish the EMF methods from the client methods. It provides support for the behaviour and functions that are common to all modelled objects.

[26]https://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/

- org.eclipse.emf.ecore.EPackage: A representation of the model object “EPackage”.
- org.eclipse.emf.ecore.EClassifier: A representation of the model object “EClassifier”.
- org.eclipse.emf.ecore.EClass: A representation of the model object “EClass”.
- org.eclipse.emf.ecore.EAttribute: A representation of the model object “EAttribute”.
- org.eclipse.emf.ecore.EReference: A representation of the model object “EReference”.

Storing Redundancy Items into RedundancyItems Class

To save the redundant elements found by the CDA tool, we implement classes to save the elements according to their type.

Figure 28 is a class diagram that illustrates the attributes, operations and relationships related to the ReundancyItems class to store the redundancy elements provided by the CDA tool. The following classes were created to represent the extracted model object accordingly. All these classes are extensions of the class *RedundancyItems*, which contains all extracted model object from “minimal-model.ecore” such as “EClass”, “EAttribute” or “EReference”:

- PrimaryAction/SecondaryAction: Which has only saved the EClass specified by “Primary/Secondary Action” and the EAttribute model object with the methods *getType* to retrieve the saved EClass and *getName* to retrieve the saved EAttribute.
- PrimaryEntity/SecondaryEntity: Which has only saved the EClass specified by “Primary/Secondary Entity” and the EAttribute model object with the methods *getType* to retrieve the saved EClass and *getName* to retrieve the saved EAttribute.
- Targets: The EClass specified by “Primary/Secondary Action” as *outgoing edge* and an EAttribute model object as *incoming edge* with “Primary/Secondary Entity”. The method *getType* retrieve the stored EClass and method *getName* retrieve the stored EAttribute.
- Contains: The EClass specified by “Primary/Secondary Entity” as *outgoing edge* and an EAttribute model object as *incoming edge* with “Primary/Secondary Entity”. The method *getType* retrieve the stored EClass and method *getName* retrieve the stored EAttribute.
- Triggers: The EClass specified by “Persona” as *outgoing edge* and an EAttribute model object as *incoming edge* with “Primary Action”. The method *getType* retrieve the stored EClass and method *getName* retrieve the stored EAttribute.
- RedundantPair: Stores the identifier of the two USs that were founded as a redundant pair. It also stored the total count of redundancy clauses of US-pair.

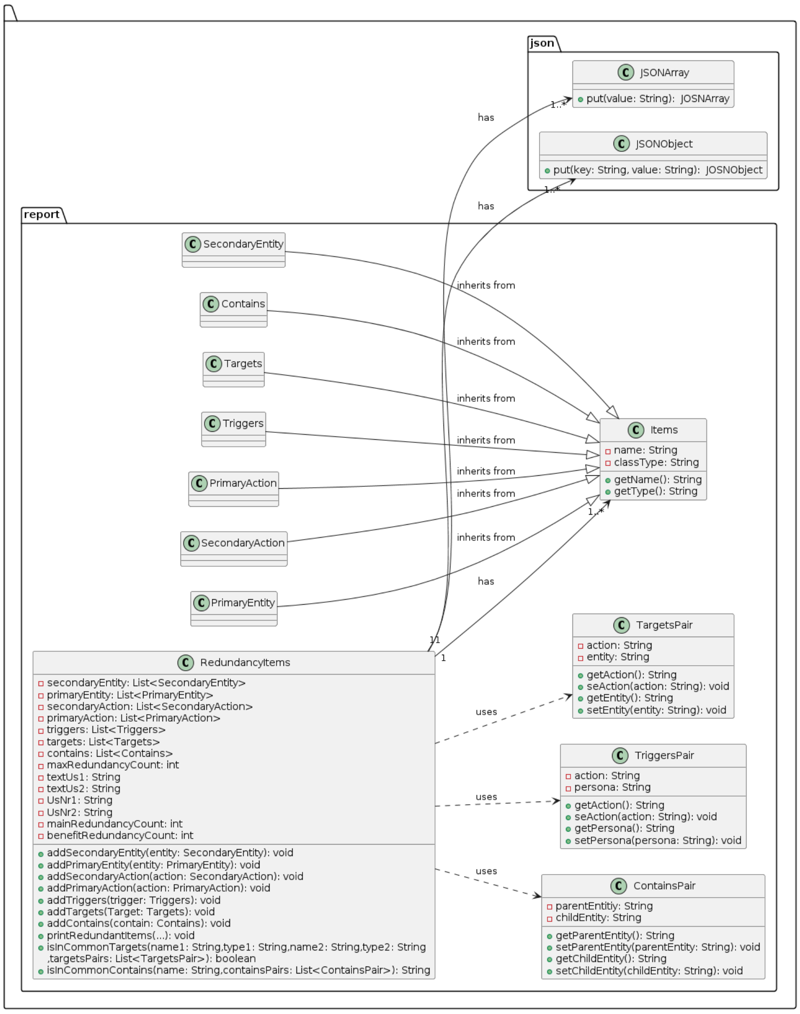


Figure 28: Class diagram for the RedundancyItems class and its relationship

- TargetsPair: Stores effective value of "*Primary/Secondary Action*" and "*Primary/Secondary Entity*" in action and entity fields accordingly.
- ContainsPair: Stores effective value of "*Primary/Secondary Entity*" as "*parent/child entity*" due to the fact that parent entity is a containment of child entity.
- TriggersPair: Stores effective value of "*Persona*" as a persona and "*Primary Action*" as an action.

The class RedundancyItems contains following methods which are important for class ReportExtractor:

- isInCommonContains: This method is designed to determine whether a given entity (specified by its name) is part of a redundant pair listed in the "Contains" array of related USs stored in a JSON file.

As input, it receives the name of the entity for which we want to check whether it exists in a redundant pair. In addition, a list of redundant pair objects that represent pairs of entities where one entity contains the other. If there is a match with the parent entity, it returns the child entity and vice versa.

- isInCommonTargets: This method is responsible for determining whether a particular action/entity is part of a redundant pair listed in the "Targets" array of related USs stored in a JSON file.

  As input, it receives the name and type of the first element, which is an action, and the name and type of the second element, which is an entity. It also receives a list of TargetsPair objects, which are pairs of common actions and entities between US-pairs.

  For each pair, it checks whether the specified names and types match either the action or entity in the pair. If a match is found, the method returns *true*, which means that the specified elements are part of a redundant pair.

- printRedundantItems: This method is responsible for creating a report on redundancy items based on the data stored in the class instance.

  As input, it takes several parameters, including a "FileWriter" to write to the textual report's file, lists of different pairs of "Targets", "Contains" and "Triggers", and a JSON-object to store the report data in JSON format, which is vital for evaluation process.

ReportExtractor Class: Methods related to Extracting Redundancy Items

To extract the redundant elements and save them in a redundancyItems object, we must first iterate through all existing *minimal-mode.ecore* files and extract the redundant item from each EPackage.

Figure 29 is a class diagram that illustrates the attributes, operations and relationship related to the extraction of redundancy items founded by the CDA tool and their storage in a RedundancyItems object.

The following methods in the ReprortExtractor class are responsible for extracting the redundancies found by the CDA tool:

- extractReports: This method orchestrates the extraction and analysis of created CDA report from a directory containing conflicted US-pairs and their associated reasons(conflict reason), and generates both text and JSON reports for further investigation and processing.

  It receives two fileWriter objects as input, one for writing textual and one for JSON reports.

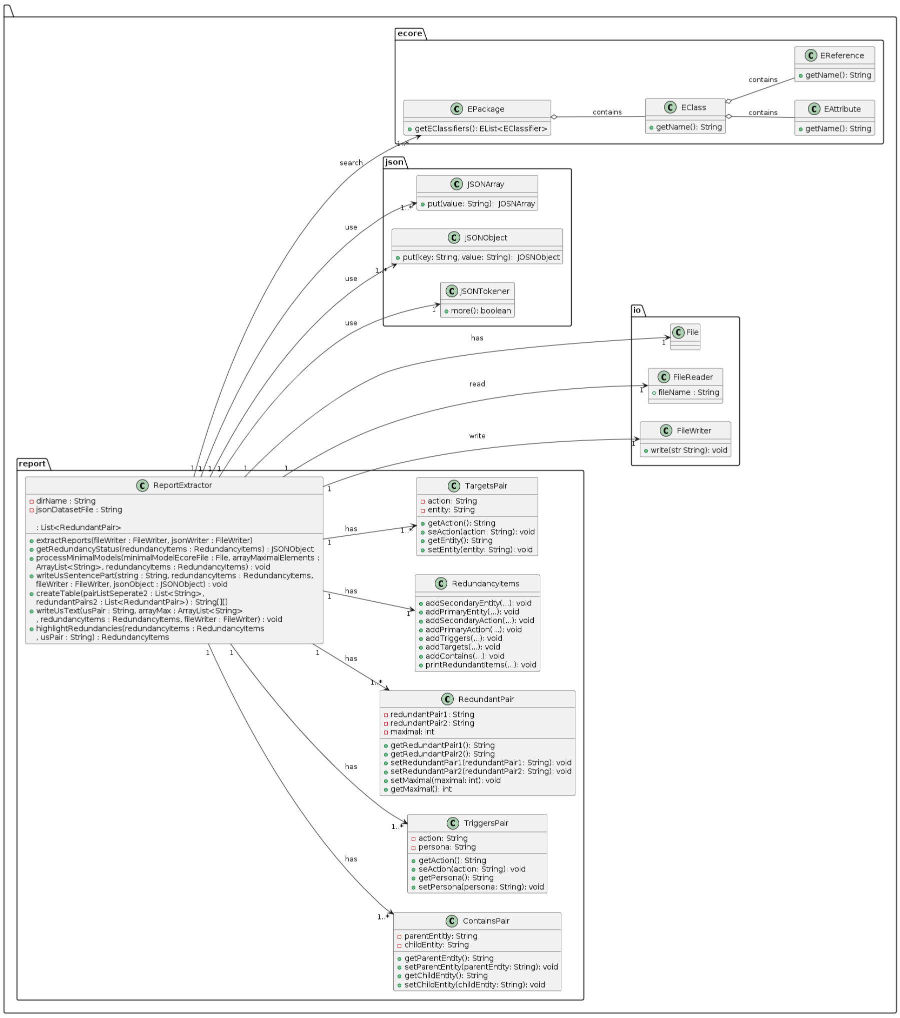


Figure 29: Class diagram for the process of extracting redundancy items

It iterates through each directory in the main directory that represents a conflict pair and uses the *checkIfReportExist* method to make sure that the current US-pair are not already proceeded and the *containsAnd* method to check whether it contains the conjunction "AND" to make sure that it is the valid US-pair name (*e.g.* "user_stroy_12_AND_user_stroy_39" is a valid directory name).

For each valid conflict pair directory, it iterates through the conflict reason directories within the current conflict pair directory and uses the *minimalEcoreExist* method to check whether a *minimal-model.ecore* file exists with reference to a conflict reason.

To identify redundancy items, it uses the method *processMinimalModels* and reads the "minimal-model.ecore" file, processes its content and iterates over the contained EPackages in order to process them further with the method *iteratePackages*, which saves all redundancy items in a RedundacyItems object.

In addition, the methods *hasEntitys*, *hasActions* and *hasTargets* are used to check whether the identified elements contain ”Primary/Secondary actions” and ”Primary/Secondary Entities” with common ”Targets” reference. If the identified elements fulfil the criteria, the redundancy pair is included in the report.

It then writes the potentially redundant USs and their clauses to the textual as well as JSON report files for further analysis.

As output, the method returns a list of RedundantPair objects containing information about identified redundancies between US-pairs.

- createOrOverwriteReportFile: The method is responsible for creating or overwriting report file. It first ensures the existence of a report file. If the file doesn't exist, it creates a new one; if it already exists, it overwrites the existing file. Finally, it returns a *FileWriter* to allow writing to the report file.

- checkIfReportExist: This method takes two parameters, namely US-pair and the list of all previously processed pairs in the CDA report directory. It returns *true* if the US-pair was found in the pairList, which means that a report with the specified pairs has already been executed and therefore does not need to be executed again.

- minimalEcoreExist: This method checks the existence of a *minimal-model.ecore* file using a conflict pair and a conflict reason generated by CDA tool.

  As input, it receives a conflict pair and a conflict reason. Using these parameters, the method constructs the file path to the minimal-model.ecore file and checks whether the file exists under the constructed path.

  If the ECore file exists, the method returns *true*, indicating that the minimal-model.ecore file exists for the examined conflict pair and conflict reason. If the file does not exist, *false* is returned.

- containsAnd: This method ensures that the folder name is identified with ”*_AND_*”, as the report generated by CDA tool is formatted like “user_story_<digit >_AND_ user_story<digit >”. It returns *true* if the examined directory contains “AND”, otherwise *false* is returned.

- processMinimalModels: This method reads a minimal model Ecore file, processes its content and iterates over the contained EPackages in order to process them further with the *iteratePackages* method.

  As parameters, it receives a File object that represents examined minimal model Ecore file, an array list in which the names of the redundant elements are stored, and a RedundancyItems object that is used to handle redundant elements.

  First, a ResourceSet and a ResourceFactoryRegistry corresponding to the minimal model Ecore file are set up and a Resource object is created from the Ecore file; the *iteratePackages* method is called for each EPackage.

- iteratePackages: This method identifies redundancies within the attributes and references of EClasses in examined minimal model, stored them accordingly and updates the RedundancyItems object.

  It takes several parameters, such as the EPackage to be iterated over, an array list in which the names of the redundant elements are stored, and a RedundancyItems object.

  It iterates through every EClassifier in the minimal package that contains EClasses and checks whether "#" is present in EClass; if this is the case, EClass has been recognised as a conflict by CDA tool.

  If an attribute is found, the class of the conflicting attribute is determined and added to the corresponding element within RedundancyItems (e.g. Primary/Secondary Action/Entity).

  Each EReference is then iterate through the EClass. Depending on the reference name, the reference is added to the corresponding element within RedundancyItems (e.g. Triggers, Targets, Contains). The method is completed once all EClassifiers within the specified EPackage have been processed.

- hasActions: This method is useful for using the data stored in RedundancyItem to determine whether certain actions are present in a list of redundant items, in order to later check whether the identified elements fulfil the criteria and the redundancy pair is included in the report.

  It checks if there's a match with name of any primary/secondary action stored in the RedundancyItems object. If match is found, it immediately returns *true*.

- hasEntitys: This method is used to determine whether certain secondary/primary entities are present in the RedundancyItems object based on their name. For each item, it checks whether there is a match with the name of a primary/secondary entity stored in the RedundancyItems object.

  If a match is found, *true* is returned immediately, in order to later check whether the identified elements fulfil the criteria and the redundancy pair is included in the report.

- hasTargets: This method uses the content of a RedundancyItems object to determine whether targets are present in a list of founded redundant elements.

  As input, it receives an array list with the names of the redundant elements and an object of the type RedundancyItems, which contains a collection of founded "Targets".

  The method checks whether there is a match with the name of any targets reference stored in the RedundancyItems object.

  If a match is found between the elements, the method immediately returns *true*, which means that at least one targets reference is present, in order to later check whether the identified elements fulfil the criteria and the redundancy pair is included in the report.

- readJsonArrayFromFile: This method provides the ability to read JSON data from a file and convert it into a JSON array object, handling cases where the file is empty or does not exist.

  It receives the file path of the JSON file to be read as input. An attempt is made to open the specified file and check whether the file is empty or does not exist. If the file exists and is not empty, it reads the JSON data from the file and creates a JSON array object from the JSON data read from the file and returns the JSON array object with the JSON data.

- getRedundancyStatus: This method add statistics like count of main/benefit/total redundancies into JSON report. It receive as input redundancyItems and as output write the count of main/benefit/total redundancies into JSON report already defined in method *highlightRedundancies*.

Methods for Extracting Data from Dataset of Backlog

To get information about USs such as text, redundant clauses in targets/contains/triggers references between US-pairs, we need to extract data from the primary input dataset.

The following methods in the ReprortExtractor class are responsible for extracting data from dataset within JSON file:

- readJsonArrayFromFile: This method receives a JSON file as input. After reading, the JSON content is tokenised, parsed into a JSON array and the parsed JSON array is returned.

- getUSsTexts: This method ensures that the text of the specified US-pair is retrieved from the JSON file and properly assigned to the RedundancyItems object for further processing. It receives a US-pair and RedundancyItems as input.

  It reads a JSON array from a file using the *readJsonArrayFromFile* method, iterates over each JSON object in the array and compares the extracted US identifier with the US identifier extracted from the input US-pair. If a match is found, the text of the first and second USs is set in the redundancyItems object.

- getCommonTargets: Used to determine overlaps in "Targets" references (pairs of actions and entities) between specified US-pairs from a JSON file.

  It receives the identifiers of the US-pair as input. After it finds the US objects, it compares each pair of entries (actions and entities) in "Targets" references of the two USs and looks for matches.

  The output is the matches, i.e. the list of common pairs of actions and entities, in "Targets" reference.

- getCommonContains: Used to determine overlaps in "Contains" references (pairs of entities) between specified US-pairs from a JSON file.

  It receives the identifiers of the US-pair as input. After it finds the US objects, it compares each pair of entities in "Contains" references of the two USs and looks for matches.

  The output is the matches, i.e. the list of common pairs of entities, in "Contains" reference.

- getCommonTriggers: Used to determine overlaps in "Triggers" references (pairs of personas and actions) between specified US-pairs from a JSON file.

  It receives the identifiers of the US-pair as input. After it finds the US objects, it compares each pair of entries (personas and actions) in "Triggers" references of the two USs and looks for matches.

  The output is the matches, i.e. the list of common pairs of personas and actions, in "Triggers" reference.

Methods for Highlighting Words using Hash Symbol(#)

In order to distinguish redundancy words between a US-pair, in text of each US, we decide to highlighting redundant words using hash symbol like " ... #<word># ... ".

The following methods in the ReprortExtractor class are responsible for highlighting redundancy words:

- writeUsText: This method reads the text of examined US-pair, highlights redundant element between them using *highlightRedundancies* method, writes the highlighted text to a file, and records information about the redundant elements.

  As input it receive a US-pair, a list of redundant pairs to store redundant pairs in it, an object redundancyItems containing stored redundancy elements and a fileWriter object used for writing output.

  It extract the identifier of the two USs from US-pair, retrieves the text of the USs from JSON file and add them to RedundancyItems, invoke the *highlightRedundancies* method to identify and highlight redundants between the USs, it writes the highlighted text of each US to the FileWriter. Finally, sets the redundant pair and count of redundancy clauses.

- highlightRedundancies: This method identifies redundancies between US-pair, applies hash symbols using *applyHashSymbols* method to highlight common items and updates the redundancy counts in the redundancyItems object.

  It takes two parameters, redundancyItems and US-pair, which represents the pair of USs to be analysed.

  It checks whether both USs contain a main clause part or whether one of them has a benefit part or whether both USs also have a benefit part.

It applies hash symbols to common elements that only occur in the part of the USs that occurs in the same part (e.g. only main or only benefit part of the USs).

In each condition, it checks if there are redundancy clauses in the main part, then persona is also highlighted using *applyHashSymbolPersona* method. It also updates the count of main/benefit/total redundancies and sets the changed text of USs. Finally, it returns the updated redundancyItems object.

- applyHashSymbols: This method is used to mark certain words within a substring with hash symbols (#) at the beginning and end to ensure that they are distinguishable and can be easily identified or processed later.

  It takes a substring in which replacements are to be made and a field of matches containing the words to be surrounded with hash symbols. First, the field of matches is sorted in descending order of length and processed accordingly to avoid adding hash symbols to unwanted clauses.

  Example 4.14. *For example, let's assume that we have "data" and "data format" as redundancy elements. If we continue first with "data" and then with "import data", "import data" will be replaced by "import #data#", which is not desired.*

- hasMoreThanFour/SixHashSymbols: These methods receive a text from the US as input. They are used to check whether there are redundant clauses in the main part of the sentence (it can be one or two clauses). If so, *true* is returned.

- applyHashSymbolPersona: This method identifies common "Triggers" references, marks them with hash symbols and returns the changed text parts together with the count of redundant triggers references.

  As input, it receives a list of common triggers between US-pair, RedundancyItems and the parts of the USs. It iterates through the list of common triggers and checks whether both elements (persona and primary action) of the triggers are present in both parts.

  It then increments the redundancy count to keep track of the count of redundant triggers. The output returned, is the text of USs containing the manipulated text parts with hash symbols and the redundancy count the in main parts.

- applyHashSymbolTargets: This method identifies common targets references between two parts of USs, marks them with hash symbols and returns the changed text parts together with the count of redundant targets references.

  As input, it receives a list of common targets between the US-pair, RedundancyItems and the parts of the USs.

  It iterates through the list of common targets and checks whether both elements (primary/secondary action and primary/secondary entity) of the targets are present.

It then increments the redundancy count in examined part of USs to keep track of the count of redundant targets.

The output returned is the text of USs containing the manipulated text parts with hash symbols and the redundancy count in examined parts (mains or benefits).

- applyHashSymbolContains: This method identifies common contains references between two parts of USs, marks them with hash symbols and returns the changed text parts together with the count of redundant contains references.

  As input, it receives a list of common contains between the US-pair, RedundancyItems and the parts of the USs. It iterates through the list of common contains and checks whether both elements of the contains are present in both parts. It then increments the redundancy count to keep track of the count of redundant contains.

  The output returned is the text of USs containing the manipulated text parts with hash symbols and the redundancy count in examined parts (mains or benefits).

Methods related to Report Divided Parts of USs

To show which part of the USs with redundancy words occur between a US-pair, we have split the individual parts of the USs and included them in the report. The following methods in the ReprortExtractor class are responsible for splitting and reporting the parts of the USs :

- splitUSsText: This method is used to split the text of two USs into separate sections based on the occurrence of redundancy clauses.

  The input is the text of the first and second US and their corresponding identifiers, a FileWriter for writing to a file and a JSON object for processing JSON data.

  It splits each US text into three parts using commas and saves the result in arrays. It iterates over parts of the first and second USs and searches for occurrences of hash symbol pairs. For each part, the number of hash symbol pairs found is counted. Finally, all parts of the records that contain hash symbols are written to a text file and a JSON file as well.

- writeUSsSentenceParts: This method facilitates the extraction and storage of highlighted USs parts from US-pair in a textual report file for further analysis.

  As input, it receives the US-pair, RedundancyItems and a FileWriter allowing the extracted USs parts to be written. It receives the text of the USs from the redundancyItems object and calls the *splitUsText* method to split the US-pair texts into USs parts with highlighted elements.

  The extracted USs parts are also written to the text and JSON report files, using the FileWriter for further processing and analysis.

Example 4.15. *Take, for example, the following US-pair:*

user_story_60: #g22# as an it staff member, I want to #know# #how# the #data# is #used#, so that I can determine what kind of basic services and functionalities are required.

user_story_04: #g22# as a data manager, I want to #know# #how# the #data# is #used#, so that I can develop more detailed usage and support scenarios with researchers.

The following sentence parts are candidates for possible redundancies between USs:

user_story_04: I want to #know# #how# the #data# is #used#

user_story_60: I want to #know# #how# the #data# is #used#

Methods related to Creating Table

The summary of potential redundancies between US-pairs is presented in a table, which makes it easy to find out which US-pairs have been identified as potentially redundant US-pairs and how many redundancy elements there are in total. The following methods in the ReprortExtractor class are responsible for creating a table at the beginning of the text report:

- writeTable: This method is used to write a table of potential redundancies between USs and the count of their total redundant clauses using *createTable* method.

  As input, it receives a File object into which the table is inserted and a list of redundancy pairs containing information about redundant clauses in US-pairs.

  It reads the existing content of the textual report file into a StringBuilder. It creates a table to display the potential redundancies between USs and the count of total redundancy clauses.

  The table headers and contents are generated based on the redundant pairs. It calculates the maximum width for each column in the table to ensure proper formatting. Finally, the table content is written to the FileWriter, followed by the existing content stored in the report's StringBuilder.

- createTable: This method prepares the content for the table in which potential redundancies between USs are displayed, taking into account the total redundancy count between each US-pairs based on the RedundantPair objects provided.

  As input, it receives a list of unique US-pairs for which the table is to be created and a list of RedundantPair objects containing information about redundant clauses between US-pairs.

It initialises a two-dimensional array containing the contents of the table. The size of the table is determined by the count of unique pairs of USs plus one for the header row and column.

It fills the header row and the first column of the table with unique pairs of US-pairs, replacing “user_story” with “us” for the purpose of brevity.

It calculates the maximum redundancy count between each US-pair by calling the method *getTotalRedundanciesFromPair*. Finally, it fills the table with the total redundancy count.

The output is a two-dimensional array representing the contents of the table, with each cell containing the maximum redundancy count between the corresponding pair of USs.

- getTotalRedundanciesFromPair: This method makes it easier to retrieve the total number of redundancies between a US-pair from a list of RedundantPair objects.

  As input, it receives a list of RedundantPair objects containing information about redundant elements in US-pairs, where the first and second USs are to be compared.

  It iterates through each RedundantPair object in the RedundantPairs list and checks for each RedundantPair object whether the examined US-pair matches as redundant US-pair.

  If a matching pair is found, the maximum redundancy number stored in this RedundantPair object is returned. If no matching pair is found, *zero* is returned, indicating that there are no redundancies between the examined US-pair.

Report Evaluation

The Evaluation class, found in the *org.henshin.backlog.code.evaluation* package, was created to assess redundancy levels in USs based on extracted JSON reports. This class includes methods that evaluate whether two USs are fully or partially redundant, focusing on various application components within these USs.

Figure 30 is a class diagram that illustrates the attributes, operations of the Evaluation class and its relationship to other classes. The class offers a detailed mechanism to evaluate redundancies through following methods:

- *evaluateRedundancyCriteria*: This method compares elements such as ”Triggers”, ”Targets” and ”Contains” in the ”Main” and ”Benefit” parts of USs. The method checks whether these clauses are fully redundant (i.e. they are identical between USs) or partially redundant (i.e. they share some clauses but are not fully identical).

  As input, it receives the JSON object for a specific US-pair.

  In particular, it checks various conditions, e.g. whether common clauses are present, whether targets or contains are empty and whether the clauses in the stories are fully or partially identical.

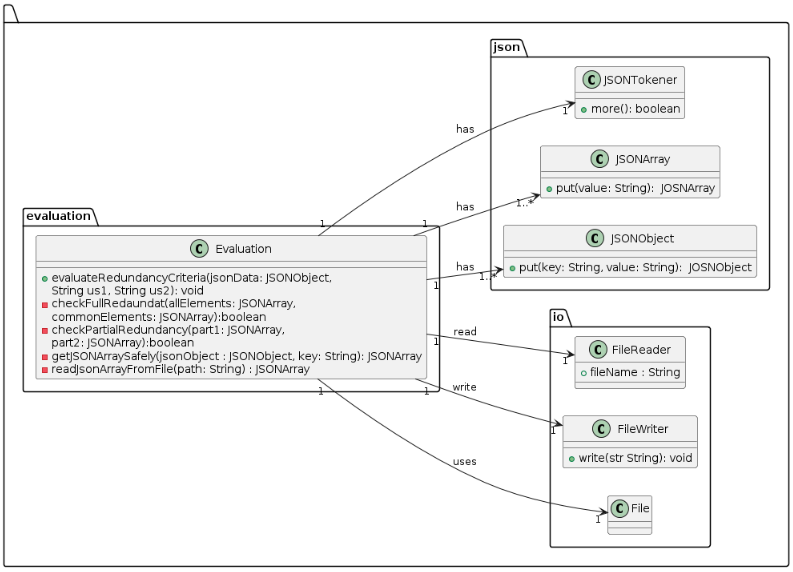


Figure 30: Class diagram related to Evaluation class

As output, the method updates the JSON report with new keys indicating the redundancy status:

Main Part Fully Redundant: A Boolean value indicating whether the ”main part” of the USs is fully redundant.

Main Part Partially Redundant: A Boolean value that indicates whether the ”main part” of the USs is partially redundant.

Benefit Part Fully Redundant: A Boolean value that indicates whether the ”benefit part” of the USs is fully redundant.

Benefit Partially Redundant: A Boolean value that indicates whether the ”benefit part” of the USs is partially redundant.

- checkPartialRedundancy: The method processes these arrays to identify partial redundancies by checking whether elements from arrays of elements match elements from another array.

  It receives the two arrays of elements as input, iterates through each element of the first input array and compares each of these elements with all elements in the second input array. A match counter records how often elements from the first input array find a match in the second input array.

  As output, method returns a boolean value. True: indicates that there is at least one matching element pair between the two JSON arrays, which means partial redundancy. False: indicates that there are no matching elements, indicating no partial redundancy between these specific parts of the USs.

Example 4.16. *Let's assume we have the following JSON arrays for two different USs:*

First JSON array: [["login", "button"], ["help", "link"]]
Second JSON array: [["logout", "button"], ["help", "link"]]

The method would determine that the second element of the first array ("help", "link") matches the second element of the second array. As there is at least one match, checkPartialRedundancy would return true, indicating partial redundancy.

- getJSONArraySafely: This method is designed to handle JSON operations safely by retrieving JSON array from a JSON object without the risk of throwing exceptions if the specified key does not exist.

  As input it receives the JSON object from which the JSON array needs to be extracted and a key corresponding to the JSON array within the provided JSON object.

  The method first checks whether the provided key exists in the JSON object. If the key exists, it retrieves the JSON array associated with that key. If the key does not exist, it returns an empty JSON array.

Example 4.17. *Let's assume that a JSON object representing US's details should contain the key for "Benefit Part", which does not exist:*
{
"Common Targets" : {
"Main Part" : [["login","button"],["help","link"]] }
}

If the "Benefit Part" in the "Common Targets" needs to be accessed that does not exist, this is handled safely with getJSONArraySafely by returning an empty array to avoid runtime errors.

- readJsonArrayFromFile: The method enables JSON data to be read from a file and converted into a JSON array object. It takes the file path as input and attempts to open the file, checking whether the file is empty or missing. If the file is available and contains data, this data is read, converted into a JSON array object and returned.

Error Handling

There were erroneous data in datasets that force us to handle them correctly. Therefore, we implement/use the following exceptions to accurately distinguish and handle them.

The following classes, which extend the *Exception* class, are used in the *org.henshin .backlog.code.rule* package and concern error handling related to the JSON entries in the dataset of the backlogs and the Ecore meta model required to create the rules based on them:

- EmptyOrNotExistJsonFile: Is triggered if the JSON file could not be found in the file system.
- ActionInJsonFileNotFound: Is triggered if the entry *Action*, which contains *Primary/Secondary Actions*, is not present in the JSON file and its absence should be reported.
- EntityInJsonFileNotFound: Is triggered if the entry *Entity*, which contains *Primary/Secondary Entity*, is not present in the JSON file and its absence should be reported.
- PersonaInJsonFileNotFound: Is triggered if the entry *Persona* does not exist in the JSON file for a specific US.
- TextInJsonFileNotFound: Is triggered if the entry *Text* does not exist in the JSON file for a specific US.
- UsNrInJsonFileNotFound: Is triggered if the entry *Us_Nr* does not exist in the JSON file for a specific US.
- EdgeWithSameSourceAndTarget: Refers to the creation of edges in graph transformation rules and is triggered if the source and target of the edge have already been created, then the duplicate edge should be avoided.
- TargetsInJsonFileNotFound: Is triggered if the entry *Targets* does not exist in the JSON file for a specific US.
- ContainsInJsonFileNotFound: Is triggered if the entry in *Contains* is not present as *Primary/Secondary Entity* in the JSON file and its absence should be reported.
- TriggersInJsonFileNotFound: Is triggered if the entry *Triggers* does not exist in the JSON file for a specific US.
- EcoreFileNotFound: Is triggered if the required Ecore meta-model file could not be found and should also be reported.

Within the *org.henshin.backlog.code.report* package, special classes that extend the *Exception* class are designed to solve problems related to the CDA report directory, which encapsulates all US-pairs together with the associated conflict reasons. These classes include:

- CdaReportDirIsEmpty: This exception is called if the CDA report directory is found but has no content.

- CdaReportDirIsNotADirectory: This exception is thrown in scenarios where the path provided for the CDA report directory is either not a directory (i.e. it is a file) or the specified path does not lead to a directory.
- CdaReportDirNotFound: This exception is triggered if the CDA report directory cannot be found within the specified path.

Limitations

There are technical limitations that are forcing us to change our implementation strategy. The following limitations should be clarified at the beginning:

- Use Eclipse version 2023-03, as Henshin version 4 cannot be installed with the latest version of Eclipse.
- Working with Java as the programming language, as the Henshin and CDA APIs are only available in the Java programming language.
- CDA API is not yet implemented to take into account conflicts and dependencies for *attributes* that are crucial for our approach. This forces us to use the CDA graphical user interface(GUI) instead of the CDA API.
- Lack of Henshin documentation regarding methods and classes, which makes it time consuming to understand the methods and make the right decision.

4.5 Test

In this section, we aim to validate certain functionalities, check the system requirements and ensure reliability and robustness of implemented classes and methods.

As a test strategy, we perform unit tests with *JUnit* version 4[27] as version 4 is more suitable and compatible with Eclipse version 2023-03.

EclEmma[28] as a code coverage tool integrated into the Eclipse IDE is used, to ensure thorough testing and coverage. By using EclEmma, we were able to systematically measure the effectiveness of our test suites and determine the test coverage for each individual class.

Configuration of Test Environment

In the main project *org.henshin.backlog*, we create a separate package called *org.henshin.backlog.test*. This package contains three Java classes namely *ReportExtractorTest.java*, *RuleCreator_Test.java*, and *EvaluationTest.java*, each of which corresponds to the Java source codes accordingly.

[27] https://junit.org/junit4/
[28] https://www.eclemma.org/

Scope of Testing

The scope of the test depends on the system requirements and the three most important implemented classes *ReportExtractorTest.java*, *RuleCreator_Test.java*, *Evaluation.java* and their methods. The implemented error handling classes are also tested.

Test Cases and their Code Coverage

We describe the specific test cases that are performed during the tests. Each test case contains a description of the test scenario, the data provided and the expected result. For refining and improving our test cases, we used a code coverage report created by EclEmma in order to increase coverage and ensure a more reliable, error-resistant application.

Table 11 shows the test cases for the RuleCreator.java class and Figure 31 shows the code coverage.

Test Case	Supplied Data	Expected Outcome	Description
testAssignCmodule	Assign a dummy ECore model file	Through an exception: *EcoreFileNotFound*	Check whether the ECore model already exists and CModule is correctly assigned
ActionNotExist (testProcessJsonFile)	JSON file with a US without “Action” entry	Through an exception: *ActionInJsonFileNotFound*	Check whether there is an entry “Action” in the related US in JSON file
EntityNotExist (testProcessJsonFile)	JSON file with a US without “Entity” entry	Through an exception: *EntityInJsonFileNotFound*	Check whether there is an entry “Entity” in the related US in JSON file
PersonaNotExist (testProcessJsonFile)	JSON file with a US without “Persona” entry	Through an exception: *PersonaInJsonFileNotFound*	Check whether there is an entry “Persona” in the related US in JSON file
TargetsNotExist (testProcessJsonFile)	JSON file with a US without “Targets” entry	Through an exception: *TargetsInJsonFileNotFound*	Check whether there is an entry “Targets” in the related US in JSON file
ContainsNotExist (testProcessJsonFile)	JSON file with a US without “Contains” entry	Through an exception: *ContainsInJsonFileNotFound*	Check whether there is an entry “Contains” in the related US in JSON file
ContainsInTargets (testProcessContainsEdges)	JSON file with a US which the entity in “Targets” entry also serves as containment	The entity in Contains should be annotated as <delete>	Check whether entity in Targets have connection as Contains, if so entity in Contains should also annotated as <delete>
TriggersNotExist (testProcessJsonFile)	JSON file with a US without “Triggers” entry	Through an exception: *TriggersInJsonFileNotFound*	Check whether there is an entry “Triggers” in the related US in JSON file
DuplicateTriggers (testProcessContainsEdges)	JSON file with a US with duplicate Triggers entries	Through an exception: *EdgeWithSameSourceAndTarget*	Verify whether there is duplicated entries in Triggers JSON-array

Test Case	Supplied Data	Expected Outcome	Description
TextNotExist (testProcessJsonFile)	JSON file with a US without "Text" entry	Through an exception: *TextInJsonFileNotFound*	Check whether there is an entry "Text" in the related US in JSON file
UsNrNotExist (testProcessJsonFile)	JSON file with a US without "Us_Nr" entry	Through an exception: *UsNrInJsonFileNotFound*	Check whether there is an entry "US_Nr" in the related US in JSON file
UndefindedEntity (testProcessContainsEdges)	Specify an entity that is not contained as "Entity" in the JSON file, but appears as "Contains"	Through an exception: *EntityInJsonFileNotFound*	Check whether the entity that appears in the *Contains* entry has already been identified as an entity
PrimaryActionNotFound (testProcessTargetsEdges)	Specify a primary action that is not contained as "Primary Action" in the JSON file, but appears as "Targets"	Through an exception: *ActionInJsonFileNotFound*	Check whether the primary action that appears in the *Targets* entry has already been identified as a primary action
PrimaryEntityNotFound (testProcessTargetsEdges)	Specify a primary entity that is not contained as "Primary Entity" in the JSON file, but appears as "Targets"	Through an exception: *EntityInJsonFileNotFound*	Check whether the primary entity that appears in the *Targets* entry has already been identified as a primary entity
SecondaryActionNotFound (testProcessTargetsEdges)	Specify a secondary action that is not contained as "Secondary Action" in the JSON file, but appears as "Targets"	Through an exception: *ActionInJsonFileNotFound.class*	Check whether the secondary action that appears in the *Targets* entry has already been identified as a secondary action
SecondaryEntityNotFound (testProcessTargetsEdges)	Specify a secondary entity that is not contained as "Secondary Entity" in the JSON file, but appears as "Targets"	Through an exception: *EntityInJsonFileNotFound.class*	Check whether the secondary entity that appears in the *Targets* entry has already been identified as a secondary entity
UndefindedEntity (testProcessTargetsEdges)	Specify an entity that is not contained as "Entity" in the JSON file, but appears as "Targets"	Through an exception: *EntityInJsonFileNotFound*	Check whether the entity that appears in the *Targets* entry has already been identified as an entity
ActionNotFound	Provision of a JSON file that contains actions in *Targets* that are not entered in primary/secondary action	Through an exception: *EntityInJsonFileNotFound*	Verify whether the action that appear in the *Targets* entry has already been identified as an action
ReadJsonArrayFromFile	Assign a dummy JSON file	Through an exception: *EmptyOrNotExistJsonFile*	Check whether the JSON file already exists
EmptyOrNotExistJsonFile	Assign an empty JSON file	Through an exception: *EmptyOrNotExistJsonFile*	Check whether the JSON file is empty

Table 11: Test cases for RuleCreator class

Table 12 shows the test cases for the ReportExtractor.java class and Figure 32 shows

RuleCreator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		0 %		n/a	1	1	3	3	1	1
processJsonFile(JSONArray)		100 %		100 %	0	10	0	53	0	1
processEntities(JSONObject, CRule, JSONObject, JSONArray, Map, String)		100 %		100 %	0	7	0	31	0	1
processActions(JSONObject, CRule, JSONObject, CNode, Map, String)		100 %		100 %	0	5	0	21	0	1
processTargetsEdges(JSONObject, JSONArray, Map, Map, String)		100 %		100 %	0	4	0	18	0	1
processContainsEdges(JSONObject, JSONArray, JSONArray, Map, String)		100 %		90 %	1	6	0	15	0	1
createRules(String)		100 %		n/a	0	1	0	7	0	1
processPersona(JSONObject, JSONArray, CRule, String)		100 %		n/a	0	1	0	4	0	1
processText(JSONObject, CRule, String)		100 %		n/a	0	1	0	4	0	1
readJsonArrayFromFile()		100 %		100 %	0	2	0	6	0	1
getJsonFileAbsolutePath()		100 %		100 %	0	2	0	4	0	1
checkEntityIsTarget(String, JSONArray)		100 %		100 %	0	3	0	6	0	1
getEcoreFileAbsolutePath()		100 %		100 %	0	2	0	4	0	1
assignCmodule()		100 %		100 %	0	2	0	5	0	1
RuleCreator(String, String, String)		100 %		n/a	0	1	0	5	0	1
processRule(JSONObject, String, CModule)		100 %		n/a	0	1	0	2	0	1
getJsonFile()		100 %		n/a	0	1	0	1	0	1
getHenshinFile()		100 %		n/a	0	1	0	1	0	1
getEcoreFile()		100 %		n/a	0	1	0	1	0	1
Total	5 of 923	99 %	1 of 66	98 %	2	52	3	191	1	19

Figure 31: Code coverage related to class RuleCreator

the code coverage.

Test Case	Supplied Data	Expected Outcome	Description
testEmptyDirectroy	Assign a dummy directory	Through an exception: *CdaReportDirIsEmpty*	Check if CDA Report directory is empty
testEmptyJSONFile	Assign an empty JSON dataset file	Through an exception: *EmptyOrNotExistJsonFile*	Check if JSON dataset file is empty
testCdaDirNotADirectroy	Assign a file instead of CDA directory	Through an exception: *CdaReportDirIsNotA Directory*	Check if assigned path is a directory
testCdaDirectroy	Assign a not readable directory	Through an exception: *CdaReportDir NotFound*	Check whether CDA directory is accessible
testFinalReportDir	Assign an empty dataset directory	Return *false* if the assigned directory is empty	Check if directory of datasets are empty
testMinimalEcoreExist	Assign a CDA directory without minimal-ecore file	Return *false* if the minimal-ecore file not found	Check whether the Ecore file already exist in CDA directory
testWriteTable	Provide a CDA report for a US-pair	A table should be created in the report file	Check whether the table for US-pairs has already been created
completeMajorElements Edge (testExtractReports)	Provision of a CDA report for US-pair with exactly one redundant "Targets" clause	Inclusion of this US-pair in the redundancy report with the entries "Action", "Entity" and "Targets"	Verifies *extractReports* method when all major elements are present in the CDA report
completeMajorElements Upper Edge (testExtractReports)	Provide a CDA report for a US-pair with at least one "Targets" clause and redundant elements such as Triggers	Generated redundancy report contains information about Secondary Entities, Secondary Actions, Targets, and Triggers	Verifies *extractReports* method when all major elements are present in the CDA report and the up per edge case is reached

Test Case	Supplied Data	Expected Outcome	Description
notCompleteMajor Elements (testExtractReports)	Provide a CDA report for a US-pair without "Targets" edge, but with action and entity	The US-pair should not be reported	Verifies the behaviour of the *extractReports* method when not all major elements are present in the input data
getBenefitPart RedundanciesElements (testExtractReports)	Provision of a CDA report for a US-pair that only contains redundancy clauses in the "Benefit" part	Check whether the count of redundancy clauses in the benefit part of the USs matches the value "Benefit Part Redundancy Clause" specified in the JSON_Report file	Verifies the behaviour of the *extractReports* method when there are redundancy elements in the benefit part of USs
getMainPartRedundancies Elements (testExtractReports)	Provision of a CDA report for a US-pair that only contains redundancy clauses in the "Main" part	Check whether the count of redundancy clauses in the main part of the USs matches the value "Main Part Redundancy Clause" specified in the JSON_Report file	Verifies the behaviour of the *extractReports* method when there are redundancy elements only in the main part of USs
getTotalRedundancy Elements (testExtractReports)	Provision of a CDA report for a US-pair that contains redundancy clauses in the "Main"and "Benefit" parts	Check whether the count of redundancy clauses in the main and benefit parts of the USs matches the value "Total Redundancy Clause" specified in the JSON_Report file	Verifies the behaviour of the *extractReports* method when there are redundancy elements in the main and benefit parts of USs
highlightPersona (testExtractReports)	Providing a CDA report for a US-pair with redundancy clauses in "Triggers" (from Persona to Primary Action) and Targets	The persona should also be marked with hash symbol if there is a redundant targets clause in the main part	Checks the behaviour of the *extractReports* method when highlighting redundant persona in USs
BenefitInBothUSs (testHlightRedundancies)	Provision of a CDA report for a US-pair where both USs have the "benefit" part	Search both the main and the benefit parts of USs and mark redundancy clauses with a hash symbol if they occur	Verifies the behaviour of the *highlightRedundancies* method when both USs have benefit parts
noBenefitInBothUSs (testHlightRedundancies)	Provision of a CDA report for a US-pair where both USs do not have the "benefit" part	Search only the main part of USs and mark redundancy clauses with a hash symbol	Verifies the behaviour of the *highlightRedundancies* method when both USs don't have benefit parts
noBenefitInUS1 (testHlightRedundancies)	Provision of a CDA report for a US-pair where only the second US have the "benefit" part	Search only the main part of USs and mark redundancy clauses with a hash symbol	Verifies the behaviour of the *highlightRedundancies* method when only the second US have benefit part
noBenefitInUS2 (testHlightRedundancies)	Provision of a CDA report for a US-pair where only the first US have the "benefit" part	Search only the main part of USs and mark redundancy clauses with a hash symbol	Verifies the behaviour of the *highlightRedundancies* method when only the first US have benefit part

Test Case	Supplied Data	Expected Outcome	Description
ContainInBenefitPart (testHighlightRedundancies)	Provide a CDA report for a US-pair with redundancy clauses of "Contains" within "Benefit" part	The entities included in Contains should be marked with hash symbol	Checks the behaviour of the *highlightRedundancies* method when highlighting redundant entities included in the Contains
ContainInMainPart (testHighlightRedundancies)	Provide a CDA report for a US-pair with redundancy clauses of "Contains" within "Main" part	The entities included in Contains should be marked with hash symbol	Checks the behaviour of the *highlightRedundancies* method when redundant entities detected in the Contains
TargetsInMainPart (testHighlightRedundancies)	Provide a CDA report for a US-pair with redundancy clauses with more than one "Targets" within the "Main" part	Founded redundancy clauses should also be marked with a hash symbol, if the main part contains more than one redundancy clause as Targets	Check the behaviour of the *highlightRedundancies* method if there are more than one redundancy clause as Targets in main part
TargetsInBenefitPart (testHighlightRedundancies)	Provide a CDA report for a US-pair with redundancy clauses with more than one "Targets" within the "Benefit" part	Founded redundancy clauses should also be marked with a hash symbol, if the benefit part contains more than one redundancy clause as Targets	Check the behaviour of the *highlightRedundancies* method if there are more than one redundancy clause as Targets in benefit part

Table 12: Test cases for ReportExtractor class

Table 13 shows the test cases for the Evaluation.java class and Figure 33 shows the code coverage.

Test Case	Supplied Data	Expected Outcome	Description
testEmptyOrNotExist JsonFile	Provision of a dummy JSON file	Through an exception: *EmptyOrNotExist JsonFile*	Check whether the provided JSON file already exists and is not empty
testJsonFile	Provision of a JSON file which is not accessible	Through an exception: *IOException*	Check whether the provided JSON file is accessible
main/benefitFull_Case1	Provide a JSON file with US-pair that have no common *Contains* entries but have common *Targets* entries in main/benefit parts	In JSON report the entry "Main/Benefit Part Fully Redundant" should be *true*	Check whether US-pair without *Contains* entries can also be fully redundant in main/benefit parts
main/benefitFull_Case2	Provide a JSON file with US-pair that have not common *Contains* entries but one US has entries as Contains in main/benefit	In JSON report the entry "Main/Benefit Part Fully Redundant" should be *true*	Check whether US-pair without common *Contains* entries can also be fully redundant in main/benefit parts

Test Case	Supplied Data	Expected Outcome	Description
main/benefitFull_Case3	Provide a JSON file with US-pair that have common *Contains* entries but one US have additional entries as Contains in main/benefit parts	In JSON report the entry "Main/Benefit Part Fully Redundant" should be *true*	Check whether USs with additional *Contains* entries from common contains can also be fully redundant in main/benefit parts
main/benefitFull_Case4	Provide a JSON file with US-pair that have exactly common *Contains* entries as well as common *Targets* in main/benefit parts	In JSON report the entry "Main/Benefit Part Fully Redundant" should be *true*	Check whether USs with *Contains* and *Targets* entries can also be fully redundant in main/benefit parts
mainBenefitFull	Provide a JSON file with US-pair that all clauses in the main and benefit parts are identical	The JSON report the entries "Main Part Fully Redundant" and "Benefit Part Fully Redundant" should be *true*	Check that all clauses in the main and benefit part are identical, main and benefit part are evaluated as fully redundant
mainPartial_Case1	Provide a JSON file with a US-pair whose entries in *Targets* are partially identical	In the JSON report, the entry "Main Part Partially Redundant" should be *true*	Check whether USs with partially redundant entries in *Targets* are evaluated as partially redundant
mainPartial_Case2	Provide a JSON file with a US-pair whose entries in *Contains* are partially identical	In the JSON report, the entry "Main Part Partially Redundant" should be *true*	Check whether USs with partially redundant entries in *Contains* are evaluated as partially redundant
benefitParial	Provide a JSON file with US-pair where only some clauses in *Targets* and *Contains* entries in benefit parts are common	In the JSON report, the entry "Benefit Part Partially Redundant" should be *true*	Check whether USs with partially common *Targets* and *Contains* entries in benefit parts are evaluated as partially redundant
mainFullBenefitPartial	Provide a JSON file with US-pair where all clauses in the main parts and some in the benefit parts are identical	In the JSON report, the entry "Main Part Fully Redundant" and "Benefit Part Partially Redundant" should be *true*	Check whether the main parts are evaluated as fully redundant if all clauses are identical and the benefit parts are evaluated as partially redundant if some clauses are identical
mainPartialBenefitFull	Provide a JSON file with US-pair where some clauses in the main parts and all in the benefit parts are identical	In the JSON report, the entry "Main Part Partially Redundant" and "Benefit Part Fully Redundant" should be *true*	Check whether the main parts are evaluated as partially redundant if some clauses are identical and the benefit parts are evaluated as fully redundant if all clauses are identical

Table 13: Test cases for Evaluation class

ReportExtractor

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
splitUsText(String, String, String, String, FileWriter, JSONObject)		100 %		100 %	0	9	0	27	0	1
reportExtractor(String[])		100 %		100 %	0	2	0	15	0	1
createTable(List, List)		100 %		100 %	0	4	0	13	0	1
getAbsoluteDirPath()		100 %		100 %	0	6	0	15	0	1
getUssTexts(String, RedundancyItems)		100 %		100 %	0	4	0	13	0	1
hasActions(ArrayList, RedundancyItems)		100 %		100 %	0	6	0	8	0	1
hasEntitys(ArrayList, RedundancyItems)		100 %		100 %	0	6	0	8	0	1
applyHashSymbols(String, String[])		100 %		100 %	0	2	0	4	0	1
getTotalRedundanciesFromPair(List, String, String)		100 %		100 %	0	6	0	6	0	1
checkIfReportExist(String, List)		100 %		100 %	0	2	0	7	0	1
hasTargets(ArrayList, RedundancyItems)		100 %		100 %	0	4	0	5	0	1
minimalEcoreExist(String, String)		100 %		100 %	0	2	0	4	0	1
getAllTargetsInBenefit(String, String, String, JSONObject)		100 %		100 %	0	2	0	8	0	1
getAllContainsInBenefit(String, String, String, JSONObject)		100 %		100 %	0	2	0	8	0	1
getAllTargetsInMain(String, String, String, JSONObject)		100 %		100 %	0	2	0	7	0	1
getAllContainsInMain(String, String, String, JSONObject)		100 %		100 %	0	2	0	7	0	1
getAbsoluteFinalReportDir()		100 %		100 %	0	2	0	4	0	1
readJsonArrayFromFile(String)		100 %		100 %	0	2	0	6	0	1
processElements(String, String, String, JSONObject)		100 %		100 %	0	3	0	4	0	1
getFinalReportDir(Path)		100 %		100 %	0	2	0	4	0	1
hasMoreThanFourHashSymbols(String)		100 %		100 %	0	2	0	2	0	1
containsAnd(String)		100 %		100 %	0	2	0	3	0	1
lambda$0(int)		100 %		100 %	0	2	0	1	0	1
lambda$2(int)		100 %		100 %	0	2	0	1	0	1
highlightRedundancies(RedundancyItems, String, List, List, List, JSONObject)		100 %		87 %	3	13	0	104	0	1
getCommonTargets(String, JSONObject)		100 %		87 %	4	17	0	69	0	1
getCommonContains(String, JSONObject)		100 %		87 %	4	17	0	68	0	1
writeTable(File, List)		100 %		87 %	2	9	0	28	0	1
iteratePackages(EPackage, ArrayList, ArrayList, RedundancyItems)		100 %		85 %	4	16	0	32	0	1
getCommonTriggers(String)		100 %		81 %	4	12	0	27	0	1
extractReports(FileWriter, FileWriter)		100 %		80 %	6	16	0	59	0	1
applyHashSymbolTargetsMain(List, List, RedundancyItems, String, String, JSONObject)		100 %		75 %	3	7	0	27	0	1
applyHashSymbolContaiansMain(List, RedundancyItems, String, String, JSONObject)		100 %		70 %	3	6	0	21	0	1
applyHashSymbolTargetsBenefit(List, List, RedundancyItems, String, String, JSONObject)		93 %		66 %	4	7	2	26	0	1
applyHashSymbolPersona(List, RedundancyItems, String, String)		100 %		66 %	2	4	0	13	0	1
processMinimalModels(File, ArrayList, ArrayList, RedundancyItems)		100 %		60 %	4	6	0	11	0	1
applyHashSymbolContaiansBenefit(List, RedundancyItems, String, String, JSONObject)		92 %		58 %	5	7	2	23	0	1
createOrOverwriteReportFile(File)		63 %		50 %	1	2	2	6	0	1
hasMoreThanSixHashSymbols(String)		85 %		50 %	1	2	0	2	0	1
main(String[])		0 %		n/a	1	1	2	2	1	1
writeUsText(String, ArrayList, List, RedundancyItems, FileWriter, List, List, List, JSONObject)		100 %		n/a	0	1	0	17	0	1
getRedundancyStatus(RedundancyItems)		100 %		n/a	0	1	0	5	0	1
writeUsSentencePart(String, RedundancyItems, FileWriter, JSONObject)		100 %		n/a	0	1	0	6	0	1
getClassName(String)		100 %		n/a	0	1	0	2	0	1
ReportExtractor(String, String)		100 %		n/a	0	1	0	4	0	1
getRefName(String)		100 %		n/a	0	1	0	2	0	1
getAttName(String)		100 %		n/a	0	1	0	2	0	1
getUsName1(String)		100 %		n/a	0	1	0	1	0	1
getUsName2(String)		100 %		n/a	0	1	0	1	0	1
getDirName()		100 %		n/a	0	1	0	1	0	1
getJsonDatasetFile()		100 %		n/a	0	1	0	1	0	1
Total	35 of 3.863	99 %	50 of 357	85 %	51	231	8	738	1	51

Figure 32: Code coverage related to class ReportExtractor

4.6 Evaluation

In this section, we address two different research questions (RQs) and provide a detailed explanation of the experimental design and execution for each question. We also present a detailed analysis of the results to provide answers to each research question(RQ), highlight their implications and provide insights into the research findings.

Research Questions

The RQs addressed in this section are as follows:

- **RQ 1**: Does our tool reliably determine the level of redundancy —either partial or full— between different parts of USs (main and benefit)?
- **RQ 2**: How does the tool's performance scale when the number of USs in a backlog increases?

Evaluation

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
evaluateRedundancyCriteria(JSONObject, String, String)		98 %		79 %	30	76	0	99	0	1
checkFullRedaundat(JSONArray, JSONArray)		94 %		70 %	3	6	0	12	0	1
checkPartialRedundancy(JSONArray, JSONArray)		100 %		100 %	0	8	0	14	0	1
readJsonArrayFromFile(String)		100 %		100 %	0	2	0	8	0	1
getJSONArraySafely(JSONObject, String)		100 %		100 %	0	2	0	1	0	1
Evaluation()		100 %		n/a	0	1	0	2	0	1
Total	8 of 613	98 %	34 of 178	80 %	33	95	0	136	0	6

Figure 33: Code coverage related to class Evaluation

Methodology

To address the RQ1: "Does our tool reliably determine the level of redundancy—either partial or full—between different parts of USs (main and benefit)?", we recap the methodology employed to analyse redundancies between USs. We utilized a systematic approach that involved several key steps:

- Data Collection: For a comprehensive assessment, we applied our approach to 19 backlog datasets presented by Mosser et al.[29]. They applied the Doccano approach to these publicly available requirements datasets[12].

  It is also worth noting that some backlog datasets (g02, g13, g17, g27) did not follow the expected sentence structure, which is why we did not include them in the evaluation results to avoid unexpected behaviour. Table 14 shows the project number of each dataset and the count of USs.

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
Project Nr.	g03	g04	g05	g08	g10	g11	g12	g14	g16	g18	g19	g21	g22	g23	g24	g25	g26	g27	g28	Total USs
Total USs	57	51	53	66	97	73	54	67	66	102	137	69	83	56	53	100	100	114	60	1458

Table 14: Project number and count of USs contained in each backlog dataset

- Identification of main and benefit parts: Each US was divided into its main part, which is the core functionality, and its benefit part, which describes the value to the persona.
- Detection of redundant clauses: Recognition of redundant clauses between a US-pair in a specific part of the USs.
- Redundancy analysis in main and benefit parts: Based on the redundant clauses identified, an automatic redundancy analysis was performed for the main and benefit parts of the US-pairs based on certain criteria indicating the level of redundancy (partial or full) in the main and benefit parts.

[29] https://github.com/ace-design/nlp-stories

Ground Truth Evaluation of USs Redundancies

Answering the first research question requires our automated system to have a reference point against which its accuracy can be measured. This reference point or "ground truth" is derived from a personal assessment and serves as a benchmark for the tool's performance.

In contrast to the tool evaluation, in the personal evaluation we did not evaluate based on labels (e.g., action, entity, persona, targets, etc.). This means that when evaluating ground truth, we evaluate partial and full redundancies based on the phrases occurring in a given parts as a one-to-one comparison.

The ground truth is the final assessment against which automated results are compared. It is derived from the personal judgement of a subject matter expert (in this case myself) using a combination of expertise, experience and specific evaluation criteria. The personal assessment is targeted at providing a reliable and accurate reference for analysing redundancies between USs. The ground truth is available in the Git repository[30].

The following criteria were taken into account in the personal assessment:

- A US-pair is assessed as fully redundant in the main and benefit parts if all phrases occurring in the main and benefit parts of at least one US overlap completely over another US and can be compared one-to-one. For example, the following US-pair is assessed as fully redundant in the main and benefit parts:
  user_story_22: "#g14# as a #consumer#, I want to #view# a #data package# online, so that I can #get# a #sense# of whether this is the dataset I want."

  user_story_24: "#g14# as a #consumer#, I want to #view# the #data package#, so that that I can #get# a #sense# of whether I want this dataset or not."

- A US-pair is considered fully redundant in the main parts, if all phrases in the main part of at least one US completely overlap. For instance:
  user_story_01: "#g14# as a #publisher#, I want to #publish# a #dataset#, so that I can view just the dataset with a few people."

  user_story_02: "#g14# as a #publisher#, I want to #publish# a #dataset#, so that I can share the dataset publicly with everyone."

- A US-pair is categorised as partially redundant in the main parts if some phrases remain distinctive between the USs. For Instance:
  user_story_09: "#g04# as a #user#, I want to be able to #view# a #map display# of the public recycling bins around my #area#."

  user_story_10: "#g04# as a #user#, I want to be able to #view# a #map display# of the special waste drop off sites around my #area#."

- The criteria mentioned were also applied to the benefit parts.

[30]https://github.com/amirrabieyannejad/Redundancy_Analysis/blob/main/Final_Reports/Evaluation_v4.xlsx

Table 15 shows the aggregation of full and partial redundancies in the main and benefit parts as ground truth. As we can see, the main parts of the US-pairs have less full

Project Nr.	G03	G04	G05	G08	G10	G11	G12	G14	G16	G18	G19	G21	G22	G23	G24	G25	G26	G27	G28	Grand Total
Main Partial		2	15	17	13	8	2	5	2	159	512	13	26	6	10	38	9	7	31	**875**
Main Full								2	1	4		1	3			2		2		**15**
Benefit Partial	2	1	2	5		6	1	6		3		2	10		1		1	2		**42**
Benefit Full		1	14	18	3	3	1	1		3		3	13		4			1		**65**

Table 15: Detail about full and partial redundancies related to main or benefit parts as ground truth

redundancies (15 cases), which means that there are 15 cases where the main parts of the US-pairs is exactly the same. This level of redundancy indicates common functionality of the US-pairs, which is a sign of overlapping features.

A much higher occurrence of partial redundancies in the main parts (875 cases) indicates that there are common elements between the US-pairs, but still enough differences to avoid a complete match.

Fewer instances of partial redundancies in the benefit parts (42 cases) indicate that the USs diverge in their benefit clauses, they are more tailored to specific project outcomes.

The higher count of full redundancies (42 cases) compared to partial redundancies in the benefit parts is interesting, as it indicates that the expected objectives of certain features are often repeated in the USs. This suggests that the project is aiming for a set of common goals regardless of the specific functionality.

Evaluation of USs Redundancies using Tool

The automated tool is designed to identify redundancies among USs based on predefined criteria. It operates by analysing the text and structure of USs to detect similarities that could indicate redundancies.

In contrast to ground truth, which is based on one-to-one comparison of phrases in specific parts of USs, tool evaluation relies heavily on specified labelling (targets, triggers, contains) annotated by the Doccano tool.

The following criteria are defined for the evaluation of US-pairs:

- Full redundancy: A pair of US is considered to have "full redundancy" in the main or benefit parts when every labelled clause in that part—comprising triggers (in the main parts), targets, and contains—are syntactically identical.
- Partial redundancy: When only some labelled clauses, such as targets, have significant overlap but are not completely identical. This means that while certain labelled clauses, such as targets, may match between USs, other labelled clauses, such as triggers or contains, may differ, meaning that the USs are not fully redundant. This scenario suggests that while USs have significant similarities, they still contain unique aspects.

Table 16 shows the aggregation of the full and partial redundancies in the main and benefit parts assessed by the tool. It is also worth noting that the numbers highlighted in red show the variation in the ground truth evaluation.

Project Nr.	G03	G04	G05	G08	G10	G11	G12	G14	G16	G18	G19	G21	G22	G23	G24	G25	G26	G27	G28	Grand Total
Main Part Partial		2	14	17	13	8	1	5	2	159	477	13	26	5	10	31	9	7	27	**826**
Main Part Full			1				1	2	1	4	35	1	3	1		9		2	4	**64**
Benefit Partial	1		1	2		6	1	3		3		2	10		1			1		**31**
Benefit Full	1	2	15	21	3	3	1	4		3		3	13		4		1	2		**76**

Table 16: Aggregation of the count of full and partial redundancies in relation to the main or benefit parts assessed by the tool

Assessment of Result: High-Level Overview Based on the datasets provided, it was found that 1,851 redundancy clauses were identified across all projects, with the highest count found in the Alfred project backlog G19 (925 clauses), indicating a significant presence of redundancies in the USs of this project.

Table 17 shows the total number of redundant clauses found in each dataset.

Project Nr.	G03	G04	G05	G08	G10	G11	G12	G14	G16	G18	G19	G21	G22	G23	G24	G25	G26	G27	G28	Grand Total
Main Part Redundancy Clauses	0	4	30	31	25	11	4	11	5	274	925	24	67	12	24	89	18	19	90	**1663**
Benefit Part Redundancy Clause	2	2	31	64	5	9	2	7	0	9	0	7	38	0	8	0	1	3	0	**188**
Total Redundancy Clause	2	6	61	95	30	20	6	18	5	283	925	31	105	12	32	89	19	22	90	**1851**

Table 17: Detail about number of redundancy clauses occurred in main and benefit parts

Assessment of Result: Aggregated Analysis We also look for an aggregate for the count of redundant clauses in both the main and the benefit parts of the USs.

In the main parts, the large count of full redundancies, especially where two clauses are present (6 cases), indicates a high degree of similarity in the functionalities described.

In the benefit parts of the USs, there is a considerable count of cases where full redundancy prevails, especially where two clauses are inserted (24 cases). The large count of full redundancies with two clauses indicates that benefits are often described in a standardised way in the different USs.

Figure 34 illustrates the aggregation of the count of redundancy clauses that occur in USs.

Assessment of Result: Detailed Insights In the dataset of backlog G19, the clause "have alfred" was repeated in all USs, indicating that "alfred" is the end product and not a specific functionality. This suggests that the quality of the USs should be considered, so that they do not contain generic information that results in a lot of redundant information (false positives) being provided.

In the main parts of the USs, 1,663 clauses were redundant ("Main Part Redundancy Clauses" in Table 17), while only 188 redundancy clauses were found in the benefit parts

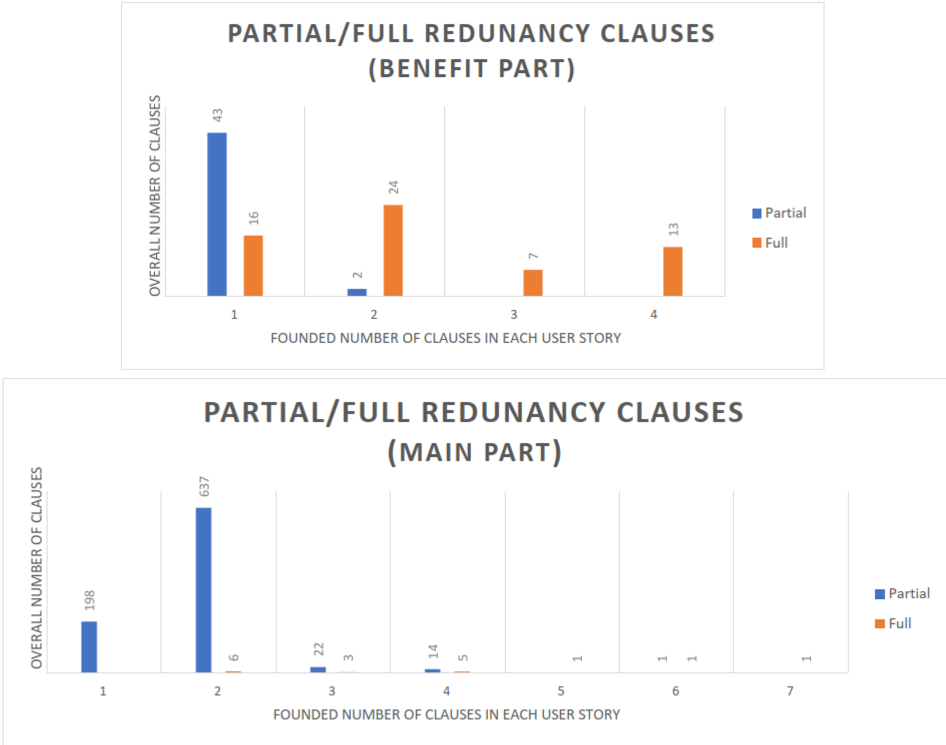


Figure 34: The aggregation between the count of redundancy clauses occurred in USs

(”Benefit Part Redundancy Clauses” in Table 17). This means that some USs are not often written in a standardised format for the description of core functions, resulting in a higher count of redundant clauses in the main parts.

Example 4.18. *In the dataset of backlog G18 (274 clauses out of 283 in the main parts), for example, this is due to the clauses “have ability” in the main parts, which are unnecessarily repeated in most USs:*
user_story_92: #g18# as a #researcher#, I want to have #the #ability# to search files by file type and format.
user_story_80: #g18# as a #researcher#, I want to have #the #ability# to attach standard metadata for behavioural observations (and video), so that my data can be searched and understood later.
Actually, the clause “have ability” in USs should be deleted and no redundancy should be possible at all:
user_story_92: #g18# as a researcher, I want to search for files by file type and format.
user_story_80: #As a researcher, I want to attach standard metadata for behavioural observations (and videos), so that my data can be searched and understood later.

There are also USs with the same functionality, but one provides more details about the functionality. In other words, one US is contained within another, and we refer to them as full redundant US-pairs, as deleting the US with less detail has no negative impact on the system. Sometimes it is necessary to merge these two USs to obtain a single detailed US.

Example 4.19. *In the dadaset of backlog G18, for example, we have two US-pairs that are marked as full redundancy between user_story_12 and user_story_11 as well as user_story_13 and user_story_11:*
user_story_12: "#g18# as a #researcher#, I want to #upload# #files# prior to having them #attached# to a #log book page# using the web interface."
user_story_11: "#g18# as a #researcher#, I want to #upload# #files# prior to having them #attached# to a #log book page#."
user_story_13: "#g18# as a #researcher#, I want to #upload# #files# prior to having them #attached# to a #log book page# using a mapped network drive."

As we can see, the user_story_11 is an incomplete version of two other USs, and deleting it has no negative impact in the system, due to the fact that its goal is achieved and fulfilled by two other USs.

RQ 1: Conclusion After comparing the ground truth result with the result provided by our tool, the following realisation was made:

- Main Part Partial: A -5.6% (negative) percentage suggests that the framework has fewer "Main Part Partial" counts than the ground truth. In this case, the framework has about 5.6% fewer "Main Part Partial" counts than expected.
- Main Part Full: A high positive percentage of 326.67% indicates that the framework has a significantly higher count of Main Part Full compared to ground truth. At 326.67%, the framework has more than three times as many main part full compared to the ground truth. In our investigation, we found that some phrases are not included as relations (contains or targets) in the USs annotated with the Doccano tool. Therefore, our tool categorised some USs as fully redundant when in reality they were only partially redundant.
- Count of Benefit Partial: -26.19% a negative percentage here indicates that the framework has fewer "Benefit Partial" counts compared to the ground truth. With -26.19%, the framework has approximately one-quarter fewer "Benefit Partial" counts.
- Count of Benefit Full: 16.92% a positive percentage indicates that the framework has more "Benefit Full" counts compared to the ground truth. With 16.92%, the framework has almost 17% more "Benefit Full" counts than the ground truth.

It is also notable that if we add founded partial and full redundancies in main or benefit parts, the both amount are identical (main part: 890 cases , benefit part: 107 cases).

This consistency indicates that the automated tool's results align with the ground truth, suggesting that the tool is reliable in detecting redundancies. It demonstrates that the tool's performance closely matches the personal assessment, reinforcing confidence in its accuracy and validity.

Performance Evaluation

To answer the RQ2: "How does the tool's performance scale as the count of USs in a backlog increases?", we conducted a series of tests to measure the time it takes the tool to process different counts of USs. This section describes the testing methodology, the results obtained and the impact on the scalability of the tool.

Test Methodology To evaluate the tool's performance, we conducted a set of experiments in which the tool processed different numbers of USs in a backlog. The tests involved the following steps:

1. Backlog Setup: We used backlogs with varying numbers of USs around—50, 70, 90, 120 and 140—to simulate different workload sizes. Each backlog contained USs with varying content, redundant elements, and complexity to represent a realistic range of cases. Table 18 shows information on the backlog data records provided for the performance test application.

US Information	Project Nr.	G04	G11	G10	G27	G19	**Grand Total**
	Number of US	51	73	97	114	137	**472**
	Number of Redundancy Clauses	6	20	30	22	925	**1003**

Table 18: Information on the backlog datasets provided for the application of the performance test

2. Tool Execution: Each part of toolchain (Rule creation, CDA tool, report creation and evaluation) was run for each backlog and the total time taken to process the entire backlog was recorded. The performance of the tool was measured by the processing time, i.e. the total time taken to process all USs in the backlog and identify redundancies.

3. Repeating Tests: To ensure reliability, each test was conducted multiple times, and the average processing time was calculated.

The test environment consisted of:

- Processor: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz (8 CPUs), 2.0GHz
- Memory: 8070MB RAM
- Display Devices: Intel(R) UHD Graphics 620, 4163 MB(Display Memory)
- Hard Disk: INTEL SSDPEKNW512G8H
- Operating System: Windows 11 Home 64-bit (10.0, Build 22631) (22621.ni_release.220506-1250)
- System Type: 64-bit operating system, x64-based processor

Table 19 shows the result of tool's performance in seconds which were conducted on a controlled environment to ensure consistency.

Test Result	Project Nr.	G04	G11	G10	G27	G19	**Grand Total**
	Rule Creation	1,32	1,49	1,53	1,63	1,67	**7,64**
	CDA Tool	13,35	55,9	63,41	80,93	1221,49	**1435,08**
	Report Creation & Evaluation	3,69	17,46	17,86	33,91	3146,49	**3219,41**
	Grand Total	**18,36**	**74,85**	**82,8**	**116,47**	**4369,65**	**4662,13**

Table 19: Information about the result of the tool's performance test, which was measured using the processing time in seconds.

RQ 2: Conclusion As we can see, there is a direct correlation between the number of USs in a backlog and the time our tool needs to process them. The results confirm that developers and project managers should expect a longer processing time when evaluating redundancies as the backlog grows.

However, the impact on processing time varies depending on the process in question. While rule creation using the Henshin API shows relatively consistent and lower processing times, other components such as the CDA tool and report creation processes show an exponential increase in processing time as the number of USs increases. This indicates that certain processes scale differently as the data load increases.

An important observation is that with a larger number of USs, the probability of finding redundant clauses between USs increases. This has a significant impact on redundancy detection and backlog management specially during processing of CDA tool.

Threats to Validity

When assessing the redundancies of USs through both personal assessment (ground truth) and the automated tool, several potential threats to validity must be considered. This section outlines the main threats to validity and describes how they were mitigated during the study.

Construct validity It refers to the extent to which the assessment measures what it is intended to measure. The following risks were identified:

- Ambiguity of criteria: If the criteria for redundancy analysis are unclear or open to interpretation, this can lead to inconsistent scores. To avoid this, we defined clear and detailed criteria for identifying redundancies in USs.
- Subjectivity in ground truth: As ground truth is based on personal judgement, subjectivity could lead to bias. To minimise this risk, we have cross-checked the reviews with other experienced reviewers.

External validity External validity refers to the generalisability of the study results to other contexts or population groups. Threats to external validity include:

- specificity of USs: If the USs used in the study are too specific or context-dependent, the results may not be transferable to other projects. To mitigate this, we included 19 backlog datasets with different USs and different project types in the analysis.

- Tool Limitation: The automated tool is designed for a specific format of USs, which limits its broader applicability. Our tool depends heavily on the specific US's format (e.g. "As a <role>, I want to ..., so that..."), which is essential for the application of the tool. Therefore, we tested the tool limited to well-formed USs and analysed its performance in a number of scenarios.

  Another limitation is that our tool is dependent on the specific type of annotation used for USs. In our case, these are annotations such as action, entity, their reference targets, triggers and contains. This means that the effectiveness of the tool depends on the accuracy and consistency of these annotations. If the annotations are incomplete or incorrect, the tool may not work as expected. In addition, the tool may not be compatible with other annotation schemes that do not use these specific labels.

Internal Validity Internal validity is concerned with whether the observed results are attributable to the factors analysed or are influenced by other variables. Potential threats to internal validity include:

- Confounding factors: External influences or unintended variables could influence the assessment of redundancy analysis. In particular, USs annotated with Doccano are a very critical impact when evaluating full and partial redundancies between USs using tool. The more phrases are covered in the relations (targets, contains), the better the score. Since our tool uses Doccano annotated USs as primary input without any changes, these discrepancies are unavoidable.
- Limitation of tool: The automated tool may not capture all aspects of redundancy analysis, especially semantically.

4.7 Conclusion

In this study, we developed and applied a comprehensive approach that combines the Doccano tool, Henshin and the CDA tool to systematically identify and report redundancies in USs within software development projects with an evaluation.

By carefully analysing 19 different backlog datasets, our method not only separated USs into main and benefit parts for nuanced examination, but also facilitated the distinction between full and partial redundancies within these parts.

Our results reveal a crucial finding: the effectiveness of redundancies analysis is significantly influenced by the quality of the USs as well as annotated USs. Well-formulated USs that do not contain unnecessary clauses (e.g. the repetition of "end product" in all USs) and a concisely annotated backlog specially in relation properties also have a major influence on the effective redundancy analysis process.

If the main parts are evaluated to be full redundant, then we have a US-pair that is functionally identical and we can merge the US-pair into one US. In the case of full redundancy in the benefit parts, this means that the US-pairs belong to the same goal and objective, to which they should be categorised for better accessibility and understanding.

A notable trend emerged from our analysis: the benefit parts are more often fully redundant than the main parts of USs. This indicates that multiple USs often strive for different functions that contribute to a common system aspect or goal.

Recognising such redundancies not only helps to consolidate functionally identical US-pairs into single, more compact USs, but also to group US-pairs together, improving accessibility and understanding of project backlogs.

In summary, our study confirms the central role of a syntactic analysis approach in detecting and managing redundancies in project backlogs, thereby contributing to the rationalisation of software development processes.

While the quality of annotated USs plays a critical role in the success of this approach, the insights gained from this research provide valuable guidance for both current practices and future research in software project management.

However, our investigation has also shown that we can only consider USs with syntactic redundancy. If they are indeed US-pairs with the same functionality but using different words and clauses to achieve the same goal, we cannot detect this with this approach.

This finding shows that the distinction between actual redundancy and mere superficial similarity can be further refined, which leads us to analyse the USs semantically.

In the next section, we therefore present a method for analysing conflicts between USs in semantic way.

5 Analysing Conflicts

In software development, especially in agile methods, conflict analysis is an important task to ensure the coherence and functionality of the system to be developed. A conflict is defined as a inconsistency that arises when two or more requirements, often encapsulated as USs, contradict each other. This section will introduce and define conflicts analysis, focusing on the concept of *content inconsistency* between USs.

The main objective of this analysis is to rationalise the software development workflow by semantically identifying conflicts between the USs within the backlog of a project.

A conflict of requirements arises when two or more USs show contradictions or inconsistencies. This can manifest itself in various forms, e.g. in the manipulation of the same resource by several USs at the same time, in overlapping functions or in conflicting conditions.

Example 5.1. *Considering following USs:*

user_story_327: "#G03# As a Staff member, I want to apply a hold, so that I can prevent progression through the workflow or other actions in the system until the issue is resolved."

user_story_328: "#G03# As a Staff member, I want to remove a hold, so that I can allow progression through the workflow or other actions in the system now that the issue has been resolved."

In this example, user_story_328 deletes a resource that is used by user_story_327. This means that user_story_327 cannot be applied at all if user_story_328 is executed first, which leads to a conflict between these USs.

In Section 5.1 we discuss about related work. In Section 5.2 we present the requirements and functional needs that serve as input for the design phase to fulfil them. In Section 5.3 we explain the design decisions of the workflow shown in Figure 35 and explain how the architecture is structured. In Section 5.4 we go into the implementation steps. In Section 5.6 we evaluate our research with two research questions (RQs) and in Section 5.7 we conclude our work.

5.1 Related Work

Kim et al. introduce a method for identifying and managing conflicts in software requirements. This method involves dividing requirements written in natural language and is supported by a specialized tool[26].

In addition, Kim et al. assert that there is a lack of methods to systematically identify and manage conflicts between requirements written in natural language within the product line context. To address this issue, they introduce a linguistic technique that detects conflicts in requirements by utilizing *goals* and *scenarios*. These goals and scenarios are expressed in natural language, specifically English.

Requirements conflicts are described as interactions and dependencies between requirements that can cause the system to function in an unintended or undesired way. An example from the mobile phone domain is given: If both the "automatic response function" and the "deny reception function" are activated at the same time, a conflict arises. If a call comes in from a number that should not be answered, "automatic response function" would conflict with the "deny reception function".

They used the goals and scenarios using an authoring structure that follows the format `action (verb) + object (object) + resource (resource)` as described by Kim et al. [25, 24] and Rolland et al. [45]. Before they described the specific requirements, they created a function dictionary in which various system functions are listed. However, they did not mention how they created the dictionary.

If action and object cause the conflict, this can lead to an activity conflict. On the other hand, a resource conflict can occur when different components try to use the same resources simultaneously, resulting in a conflict.

Kim et al. used two methods for detecting conflicts: *syntactic* and *semantic*. The syntactic method identifies potential conflicts among requirements based on predefined conditions. The semantic method then focuses on detecting actual conflicts by analysing the candidate conflicts.

Their approach is not well suited to agile development projects, as these projects usually use the US notation as a standard structure. In addition, they have not considered how elements such as objects, resources and verbs relate to each other, which makes it more difficult to recognise conflicts. They also overlook the importance of containment objects and resources when analysing conflicts.

Lambers et al. present a static CDA technique for graph transformations that can identify conflicts and dependencies at multiple levels of granularity in software development. This new technique significantly improves the speed of fine-grained CDA computations and extends them with coarse-grained results, providing usability benefits in various use-cases. It is particularly useful for analysing interactions in complex and dynamic object structures, such as feature interactions in software product line development [30].

Lambers et al. discuss the use of CDA in different areas of software development and the requirements for the granularity of these applications. They categorise the use-cases into:

- Software System Design and Analysis: Here, behavioural models such as activity models and live sequence diagrams are checked using rule-based specifications.
- Model-Driven Engineering: This involves model transformations in which the CDA helps to recognise and explain conflicts and dependencies between transformation specifications.
- Optimization of Rule-Based Computations: CDA is used to identify conflicting or interdependent rule pairs in order to optimise calculations and avoid or postpone backtracking.

However, the application of GT in USs to analyse conflicts is still unexplored.

5.2 Requirements

In order to accomplish the analysis of conflicts in USs we try to address following functional requirements:

- As a user, I want to perform semantic analysis on user stories within a specified project backlog, so that I can identify and address conflicts effectively.
- As a user, I want a report on the US-pairs that are conflicting in the main parts, so that I can change them if needed.
- As a user, I want to apply a filter to the conflict report to show US-pairs that have the same resource (as entity) with different verbs (as action), so that the verbs are semantically contradictory (e.g. one US deletes a resource that another US is using/deleting or one US creates a resource that another US prohibits).
- As a user, I want to analyse conflicts between US-pairs with different resources, where one resource acts as a child of another resource, so that this type of relationship can also be analysed.

- As a user, I want to mark the container relation with a hash symbol (#) and display found resources as a conflict element (as a noun) so that I can better recognise whether the contained entity is also recognised as a potential conflict.

- As a user, I would like to have a conflict report that shows founded US-texts pairwise and adds a hash symbol (#) at the beginning and end of conflicting verbs and the noun (as a resource) as a marker, so that I can better recognise the verbs and the noun that conflict in US-pairs.

- As a user, I want to see how many conflicted US-pairs have been founded in the main parts of the USs within a backlog, so that I can summarise conflict US-pairs founded on this basis for further statistical purposes.

- As a user, I want a table at the top of the conflict report that lists the US-pairs in conflict, so that I can quickly see all the US-pairs that have been founded.

To judge the operation of the system, we define following non-functional requirements:

- Testability: The system should support automated test procedures to ensure that semantic analysis and conflict detection work correctly. It should include comprehensive test cases covering different scenarios, including edge cases, to verify the accuracy and reliability of conflicts analysis.

- Documentation: The system should include detailed documentation covering all aspects of functionality and setup.

- Performance: The system should perform the conflicts analysis within a reasonable time frame, even with large project backlogs. It should be optimised so that it can process large volumes of data without any significant loss of performance.

- Scalability: The system should be scalable to handle an increasing count of USs and larger project backlogs.

5.3 Design

This section describes the operational flow and architectural considerations that underpin the framework.

In our project, we initially considered using the Henshin API and CDA to implement conflicts analysis between USs. The target was to show that GT can effectively analyse conflicts between USs. However, after further evaluation, we realised that using Henshin and CDA was not the most efficient approach for our needs.

While graph transformations are useful and suitable for our task, they introduce additional steps into the conflicts analysis process that can be time-consuming and complex. Our target was to streamline the process and get results faster, which required a simpler approach.

To overcome these challenges, we have developed our own tool inspired by Henshin and CDA tools. Our tool focusses on the core task of conflicts analysis but is designed to be faster as it does not require any further steps (e.g. creating rules, applying CDA to rules and interpreting the delivered reports). In this way, we can achieve similar results in a more straightforward and faster way.

Design Overview

To address the requirements specified in Section 5.2, our system used the backlogs labelled with Doccano tool[31] generated by Mosser et al. as the primary input[6].

To perform the conflicts analysis, a one-time preparation phase is required to create a database of action annotations and verbs using the VerbNet verb classification. Once this has been done, the conflict analysis can be applied to the USs using this database and specific criteria.

Once the conflicts analysis has been applied, a comprehensive report is created that contains information regarding conflicted pairs in both a textual and a tabular form. Finally, a conflict assessment phase is initiated for further statistical purposes.

Figure 35 illustrates how each step in this sequence is interconnected, with the output of one step feeding directly into the next. This diagram effectively demonstrates the toolchain and process workflow, highlighting how each step transforms artefacts and contributes to the overall objective of conflict detection.

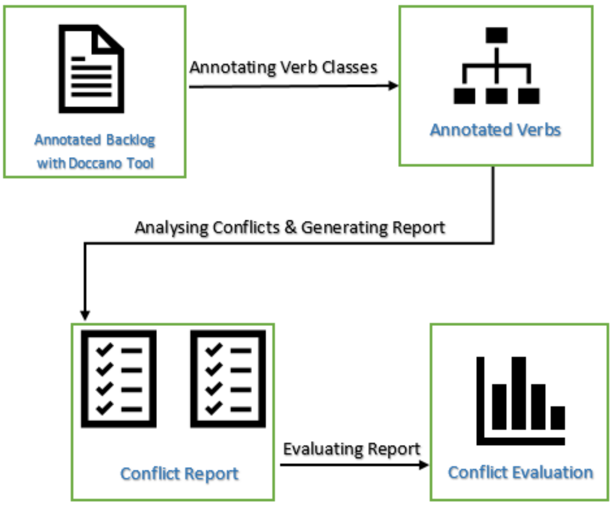


Figure 35: Step-by-step visualisation of the toolchain and its inputs and outputs

One-Time-Phase: Creating Database for annotated Verbs In this phase, each verb should be classified using the VerbNet[32] classification. Finally, each verb class annotated into four categories, namely *Create, Delete, Preserve, or Forbid* (from now on) called *action-annotation.*

[31] https://github.com/ace-design/nlp-stories
[32] https://verbs.colorado.edu/verbnet

To validate the action annotations, a personal judgement is made for each verb by three evaluators, so that each person reviews the action annotations for each verb and comments their own action annotation. We then collect all the personal judgements and combine the action annotations for each verb.

Conflict Analysis and Extraction of text reports Creating a text report aims to find conflicts between USs and highlight important information, such as identifying potentially conflict pairs, the conflict reason, the resource (as noun) affected by actions (as verbs) causing the conflict, the texts of the main parts with the affected elements marked with # and a tabulation of the potentially conflicting pairs.

Evaluating the reports Once we have created the reports, we can now assess the correctness of the US-pairs reported as conflicts, i.e. whether the reported US-pairs really cause a conflict.

Software Architecture

In this section, we present the basic structures of our workflow and the discipline of creating such structures. Each structure comprises software elements, relations among them, and properties of both.

- Annotated USs with Doccano Tool[33]: Mosser et al. used publicly available requirements from Dalpiaz et al.[13] consisting of 19 product backlogs and 1,458 USs. The dataset is a raw archive of 19 text files, each containing one US per line.

  As there were no public expert-based annotations, Mosser et al. manually annotated the dataset using the Doccano tool for *Named Entity Recognition*. Labels included persona, action, entity, benefit part and relations such as triggers, targets, and contains based on their domain meta-model.

  As artefact we receive a graph-based model with JSON format, which represents the refined and annotated dataset for the recognition of *entities*, *actions*, *personas* and *benefits* of USs [40].

- Eclipse as IDE[34]: Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base work workspace and an extensible plug-in system for customizing the environment.

- VerbNet as Verb Lexicon Resource[35]: VN is the largest on-line network of English verbs that links their syntactic and semantic patterns. It is a hierarchical, domain-independent, broad-coverage verb lexicon with mappings to other lexical resource,

[33]https://github.com/doccano/doccano
[34]https://eclipseide.org/
[35]https://verbs.colorado.edu/verbnet/

such as WordNet[36], PropBank [37], and FrameNet [38].

VerbNet is organized into verb classes extending Levin (1993) classes through refinement and addition of subclasses to achieve syntactic and semantic coherence among members of a class. Each verb class in VN is completely described by thematic roles, selectional preferences of the arguments, and frames consisting of a syntactic description and a semantic representation with subevent structure patterned on the Dynamic Event Model of Pustejovsky and Moszkowicz and Pustejovsky[27].

- USPartExtractor Class: This class is part of the *org.backlogconflict.code .preparation* package and is a key component of the software architecture designed for transforming primary input datasets in the specific JSON format. It separates elements based on their occurrence in both the main and benefit parts of the USs.

  This process simplifies subsequent tasks for editing, analysing and conflict resolution by providing a clear structure for the USs and their components. The separation of main and benefit parts and the assignment of unique identifiers improves the manageability and traceability of USs within the system.

- Action Annotation Reference Database: This database is essential for identifying conflicts between USs, especially conflicts arising from actions over common entities (as resources). To achieve this, we categorise verbs into four different groups namely *Preserve, Delete, Create, and Prohibit.*

  The main purpose of the action Annotation reference database is to facilitate the translation of actions (in the form of verbs) found in USs into corresponding action-annotations. This process involves several important steps:

  1. Collection of Actions: We collect all actions (represented as verbs) from existing datasets and compile them into a CSV file. This file serves as comprehensive reference database.
  2. Contextual Translation: Each verb in the CSV file is translated into the corresponding action-annotations related to its VerbNet class.
  3. Personal Judgement: To validate the action annotations, three evaluators individually assess each verb. Each evaluator reviews the annotations and provides their own comments. We then gather these individual assessments and combine them to finalize the action annotation reference database for each verb.

- VerbFinder Class: The `VerbFinder` class is an essential component within the *org.backlogconflict.code.preparation* package, designed to interface with the action annotation database and facilitate the process of mapping verbs to their corresponding action annotations.

[36]https://wordnet.princeton.edu/
[37]https://propbank.github.io/
[38]http://framenet.icsi.berkeley.edu/

- ActionsAnnotationsCreator class : This class is a component within the *org.henshin.backlogconflict.code.preparation* package. Its primary function is to improve the extraction of Parts of US process by incorporating action annotations into the JSON dataset.

  The class adds entries that consist of a set of triples: "action", "entity", and "action-annotations". These action-annotations are sourced from the reference database, which ensures that conflict detection and resolution are consistent and accurate. The matching process is done using the VerbFinder class.

- ReportMaker Class: This class developed within the *org.backlogconflict .code.report* packages, its primary function is to identify conflicting US-pairs based on specific criteria and generate comprehensive reports on these conflicts. It performs the following key tasks:

  1. Identification of Conflict US-pairs: The class analysis USs to identify pairs that conflict based on predefined criteria. These criteria including conflicting actions, or inconsistencies in the USs.
  2. Detailed Conflict Reporting: Once conflicts are identified, the class generates detailed reports. These reports contain essential information such as the affected entity(as resource), potential conflicted actions, conflict reason, and the texts of main parts of the USs with marked elements with hash symbol (#).
  3. Tabular Summary: In addition to detailed conflict descriptions, the class also produces a tabular summary of all identified conflict pairs in a backlog, providing a quick overview of the conflict US-pairs.

Figure 36 shows the architectural composition, highlighting the integral components and their user interface and artefacts. Regarding conflict, some definitions are clarified:

Definition 5.1 (**Customized User Story**). *In order to apply conflict analysis to the backlog, a customized user story is defined, which consists solely of the main part that collectively describes what the user wants and the consequences of this need for the resources.*

- *The main part is essential as it clearly and concisely summarizes the persona, the intended functionality, and the resources required to perform the action. This part usually follows the format: "As a [persona], I want [actions over entities]."*

  In this customized US, the intended functionality, which describe the action that the persona wants to perform or the function they need, will be translated into four annotations: "create," "delete", "preserve", or "forbid". These annotations serve to standardize the actions for conflict analysis:

  - ***Create:*** *This action-annotation describes the introduction or addition of a new entity within the system. For example, the action "apply" in US "As a staff member, I want to apply for a hold." is annotated with "create" action.*

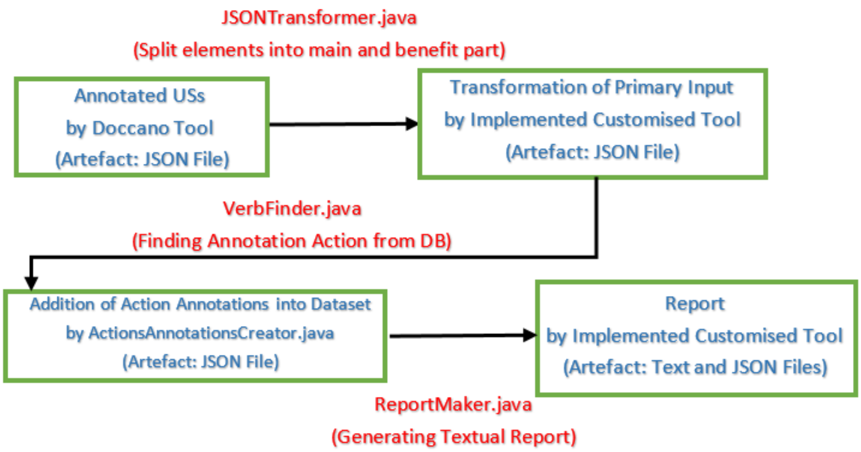


Figure 36: Design phases

- ***Delete:*** *This action-annotation indicates the removal or elimination of an entity from the system. For example, the action "remove" in US "As a staff member, I want to remove a hold." is annotated with "delete" action.*
- ***Preserve:*** *This action-annotation involves safeguarding, or using an existing entity without alterations. For example, the action "browse" in US "As a researcher, I want to browse through files in a collection." is annotated with "preserve" action.*
- ***Forbid:*** *This action-annotation specifies prohibiting certain actions on an entity. For example, the action "restrict access" in US "As a collection curator, I want to restrict access to my collection or items to duke IP addresses." is annotated with "forbid" action.*

Specifying the resources required to perform the action helps with planning and resource allocation, ensuring that the development team is aware of the tools, technologies, and time required. This includes identifying all entities involved in the actions described and their relationships.

In other words, with respect to the action, we translate it into the aforementioned action-annotations for conflict analysis. This translation standardizes the actions, making it easier to identify and resolve conflicts between USs.

It is worth noting that in this form of US, the benefit part is not considered as part of the structure of USs. The focus is solely on the actions and resources, simplifying the US to its core components necessary for conflict analysis.

Definition 5.2 (**Conflict**). *Conflict refers to situations where two USs try to:*

- *delete a resource which another US are using*

- *delete a resource which another US also wants to delete*
- *create a resource which another US prohibits*

Notation. Lowercase identifiers refer to single elements, and uppercase identifiers denote sets.
A user story is a 1-tuple $us = \langle m \rangle$ where:

- *A main m is define a 6-tuple:*

  $m = \langle p, A, E, Tr, Ta, Co \rangle$

  where:

  - p *is the persona.*
  - $A = \{a_1, a_2, ...\}$ *is a set of actions.*
  - $E = \{e_1, e_2, ...\}$ *is a set of entities.*
  - $Tr = \{(p_1, a_1), (p_2, a_2), ...\}$ *is a set of trigger references, each begin a pair of persona and action.*
  - $Ta = \{(a_1, e_1, R_1), (a_2, e_2, R_2), ...\}$ *is a set of target references, each begin a triple of action, entity, and action-annotations* R.
  - $Co = \{(e_{c1}, e_{c2}, R_{c1}), (e_{c*}, e_{c*}, R_{c2}), ...\}$ *is a set of contain references, each begin a triple of two entities and action-annotations* R.
  - $R = \{preserve, create, delete, forbid\}$ *are the annotations applied to actions.*

To denote that a syntactic operator, we add the subscript "syn"; for instance, $=_{syn}$ is syntactic equivalence which introduced by Lucassen et al. [35].
Consider two USs:

$us_1 = \langle m_1 \rangle$ *where* $m_1 = \langle p_1, a_1, e_1, tr_1, ta_1, co_1 \rangle$

$us_2 = \langle m_2 \rangle$ *where* $m_2 = \langle p_2, a_2, e_2, tr_2, ta_2, co_2 \rangle$ *and* $co_2 = (e_{c1}, e_{c2}, R_{c1})$

us_1 *causes a conflict if:*

1. *The entity e_1 is an exact redundant of entity e_2, formally:*
   $isRedundant(e_1, e_2) \leftrightarrow e_1 =_{syn} e_2$ *and one of the following conditions holds:*

   (a) $ta_1 = (e_1, a_1, \text{"}preserve\text{"})$ *and* $ta_2 = (e_2, a_2, \text{"}delete\text{"})$
   (b) $ta_1 = (e_1, a_1, \text{"}create\text{"})$ *and* $ta_2 = (e_2, a_2, \text{"}forbid\text{"})$
   (c) $ta_1 = (e_1, a_1, \text{"}delete\text{"})$ *and* $ta_2 = (e_2, a_2, \text{"}delete\text{"})$

2. *The entity* e_1 *is an exact redundant of* e_{c1}*, formally:* $isRedundant(e_1, e_{c1}) \leftrightarrow e_1 =_{syn} e_{c1}$ *and one of the following conditions holds:*

   (a) $ta_1 = (e_1, a_1, "preserve")$ *and* $co_2 = (e_{c1}, e_{c2}, "delete")$

   (b) $ta_1 = (e_1, a_1, "create")$ *and* $co_2 = (e_{c1}, e_{c2}, "forbid")$

   (c) $ta_1 = (e_1, a_1, "delete")$ *and* $co_2 = (e_{c1}, e_{c2}, "delete")$

To comprehensively assess conflicts, it is important to consider not only the textual content but also the functional relevance of each action over entities within the USs. By categorizing actions into four groups, conflict that may not be immediately apparent through a simple text comparison can be uncovered, thereby reducing time consumed in finding conflicts manually.

Design Phases

To provide a comprehensive overview of the design phases, this section explains each step of the process, from initial setup to final evaluation, using practical examples.

Listing 9 is the pseudocode that describes how to analyse the conflicts in a particular abstraction and explains the logic and sequence of steps clearly and comprehensibly.

```
funct conflict-analysis(dataset: Dataset)
  var ConflictPairs ← Null
  for each US_a in dataset do
    var MainPart_a, BenefitPart_a ← DivideIntoMainAndBenefitParts(US_a)
    var AN_a ← GetActionAnnotationFromDB(MainPart_a)
    dataset ← AddActionAnnotationIntoDataset(AN_a)
    var Entity_a ← GetEntity(MainPart_a)
    for each US_b in dataset do
      var MainPart_b, BenefitPart_b ← DivideIntoMainAndBenefitParts(US_b)
      var AN_b ← GetActionAnnotationFromDB(US_b)
      dataset ← AddActionAnnotationIntoDataset(AN_b)
      var Entity_b ← GetEntity(US_b)
      var ContainEntity_b ← GetEntityFromContains(US_b)
      if (HasConflict(AN_a, AN_b) == True) then
        if(Entity_a == Entity_b) or (Entity_a == ContainEntity_b) then
          ConflictPairs += new AddConflictPair(US_a,US_b)
  Return ReportAllConflicts(ConflictPairs)
```

Listing 9: The pseudocode describes how to analyse the conflicts in a specific abstraction

Data Structure

As primary input, we receive a graph-based model generated by the Doccano tool, which represents the refined and annotated dataset for the recognition of *entities*, *actions*, *persona* and *benefits* of USs [6].

The datasets have the JSON format, the structure of which is very important in the Java classes *USPartExtractor*, and *ActionsAnnotationsCreator*. Therefore, understanding the JSON format provided is needed for the next steps.

Each JSON file for a backlog dataset contains a JSON-array in which each US entry is defined as a JSON-object. Listing 10 illustrates the format used for the US JSON object.

```
{
 "id":...,
 "text":"...",
 "entities":[
  {"id": ...,"label":"PID","start_offset": ... ,"end_offset": ...},
  {"id": ...,"label":"Persona","start_offset": ... ,"end_offset": ...},
  {"id": ...,"label":"Entity","start_offset": ...,"end_offset": ...},
  {"id": ...,"label":"Action","start_offset": ...,"end_offset": ...},
  {"id": ...,"label":"Benefit","start_offset": ...,"end_offset": ...}
 ],
 "relations":[
  {"id": ...,"from_id": ...,"to_id": ...,"type":"triggers"},
  {"id": ...,"from_id": ...,"to_id": ...,"type":"targets"},
  {"id": ...,"from_id": ...,"to_id": ...,"type":"contains"}
 ]
}
```

Listing 10: The JSON format of each US JSON object in JSON file

A US JSON object in this JSON format contains the following elements:

- ID: Is the actual US identifier
- Text: Is the complete text from US
- Entities: is an array of JSON objects with the following elements:
  - ID: Serves as a reference number for the entry
  - Label: Is the name of the entry, which can be ”PID”, ”Persona”, ”Entity”, ”Action” or ”Benefit”
  - Start_offset: Refer to the position of the first character of the word within the entire text
  - End_offset: Is the position immediately after the last character of the word within the entire text
- Relations: Is an array of JSON objects with the following entries:

– ID: Serves as a reference number for the entry
– From_ID: Is the entry ID of the first element that occurs in the relation
– To_ID: Is the entry ID of the second element that occurs in the relation
– Type: Is the type of relation, which can be ”Triggers”, ”Targets”, or ”Contains”

As relation, each *Persona* is linked to each *Action* as `Triggers` relation, each *Action* is linked to each *Entity* as `Targets` relation and each *Entity* is linked to each *Entity* implying a `Contains` relation.

Step 1: Split Elements into Main and Benefit Parts

Since the original JSON format does not distinguish between the elements belonging to the main or the benefit part, this step brings relief and more accuracy, which is useful when processing elements in certain parts of the US.

For this reason, we use the class `USPartExtractor`, which creates a new JSON format in which all elements referring to a US JSON object are extracted from the primary input and stored in the appropriate part of US.

Listing 11 illustrate the format of JSON file generated by USPartExtractor class.

```
[
	{
		"US_Nr": "...",
		"US_ID": "...",
		"Text": "...",
		"PID": "...",
		"Main": {
			"Entity": [...],
			"Action": [...],
			"Persona": [...],
			"Contains": [...],
			"Text": "...",
			"Triggers": [...],
			"Targets": [...],
		},
		"Benefit": {
			"Entity": [...],
			"Action": [...],
			"Contains": [...],
			"Text": "...",
			"Targets": [...],
		},
		"Mix": {
			"Targets" : [...],
			"Contains": [...]
		}
	},
```

```
        ...
]
```

Listing 11: The new JSON format generated by USPartExtractor class

If the elements in a relationship come from both the main and the benefit part, we store them in the `Mix` JSON object. As the conflict in the benefit part is not taken into account in our approach, we exclude the elements that originate from the benefit and the mix part.

Step 2: Adding Action Annotations

In order to facilitating conflict analysis, we first need to translate each verb that refers to a US into four categories, namely 'preserve', 'create', 'delete', and ' forbid'. For this reason, we collect all verbs and the corresponding action annotations in a database called "action annotation database".

Secondly, we use the `VerbFiner` class to search the database for the verb associated with a US and return the corresponding action annotation/s. Thirdly, for each US JSON object, we add a JSON object labelled 'Action Annotations' to the Main JSON object using the `ActionsAnnotationsCreator` class, which consists of two elements:

- Targets Action Annotation: This is an array of 'targets', where each element is a triple of 'action', 'entity' and additionally one or more 'action annotation' separated by '`;`'.
- Contains Action Annotation:This is an array of 'Contains', where each element is a triple of 'action', 'entity' and additionally one or more 'action annotation' separated by '`;`'.

Example 5.2. *Listing 12 illustrates an example of adding elements in two US created by the ActionsAnnotationsCreator class, which contains all the elements we need for the conflict analysis.*

In this example, we see that in user_story_13 the verb 'apply' in "Target Action Annotation" is translated into two action annotations, namely 'create' and 'preserve' (third elements of the array), and the verb 'remove' in "Target Action Annotations" of user_story_14 is translated into the action annotation 'delete'.

```
[
        {   "US_Nr": "user_story_13", "Benefit": {...},
                "US_ID": "user_story_327", "Text": "..., "PID": "#G03#",
                "Main": {"Entity": ["Hold"], "Action": ["Apply"],
            "Persona": ["Staff member"], "Contains": [],
            "Text": "#G03# As a Staff member, I want to Apply a Hold",
            "Triggers": [["Staff member","Apply"]],
            "Targets": [["Apply","Hold"]],
```

```
            "Action Annotations": {"Target Action Annotations": [
                        ["apply", "hold", "create;preserve"]],
                "Contain Action Annotations": []
            }},
        "Mix": {}
        },
                {"US_Nr": "user_story_14", "Benefit": {...},
        "US_ID": "user_story_328", "Text": "...", "PID": "#G03#",
                "Main": {"Entity": ["Hold"], "Action": ["Remove"],
            "Persona": ["Staff member"], "Contains": [],
            "Text": "#G03# As a Staff member, I want to Remove a Hold",
            "Triggers": [["Staff member","Remove"]],
            "Targets": [["Remove","Hold"]],
            "Action Annotations": {"Target Action Annotations": [
                        ["remove","hold","delete"]],
                "Contain Action Annotations": []
                        }},
        "Mix": {}
    }
]
```

Listing 12: Example of adding action annotations to the individual verbs

Step 3: Analysing Conflict and Creating Report

After we have inserted the action annotation into the US JSON object, we can now analyse the conflict and simultaneously report the founding conflict by using the `ReportMaker` class. The steps involved in this process are as follows:

1. Create text and JSON formats for reporting: To report a set of datasets in a single file, we create two 'consolidated conflict reports', one in text format and one in JSON format. We also create two types of reports for individual datasets (one in text format and one in JSON format). It is also worth noting that creating the JSON format facilitates the conflict analysis evaluation process.

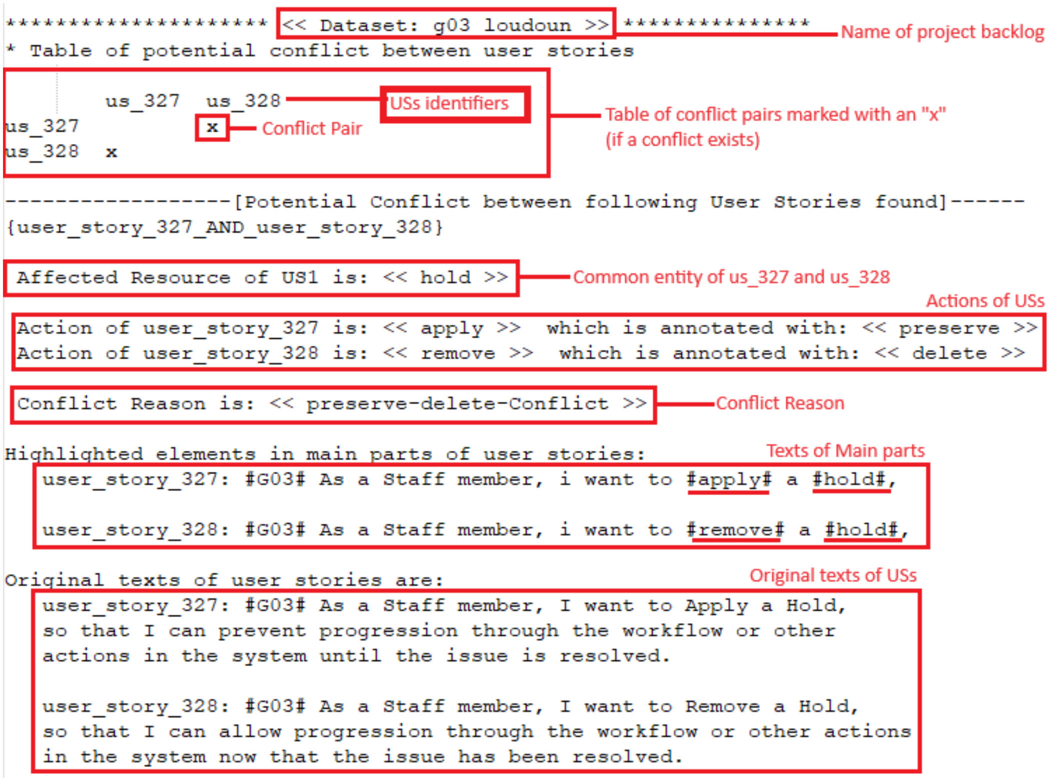
2. Iterate through each dataset and analysing conflicts: We iterate over the USs to identify conflicts by comparing the action annotations and applying defined criteria between one US and another.

   If there is a potential conflict pair, we store this pair and its information such as US_ID, conflict reason, text of USs, affected entity, conflict actions, etc. to report them.

3. Create summary table(only applicable for the text report): A summary table of potential conflicts between USs is created at the top of the textual report files and marked with an 'x' if a conflict exists.

4. Generate and write reports: It writes the conflict information, including US_ID, actions, conflict reason and elements highlighted with '#' in the main parts, to the report files.

Example 5.3. *In this example, we apply the conflict analysis to the project* `G03_loudoun`*. As a result, we get the generated text report consisting of a conflict pair between user_story_327 and user_story_327 as shown in Listing 13, and the part of the report has been marked and noted.*



```
******************** << Dataset: g03 loudoun >> ***************
* Table of potential conflict between user stories

          us_327   us_328
us_327             x
us_328    x

-----------------[Potential Conflict between following User Stories found]------
{user_story_327_AND_user_story_328}

 Affected Resource of US1 is: << hold >>

 Action of user_story_327 is: << apply >>  which is annotated with: << preserve >>
 Action of user_story_328 is: << remove >>  which is annotated with: << delete >>

 Conflict Reason is: << preserve-delete-Conflict >>

Highlighted elements in main parts of user stories:
   user_story_327: #G03# As a Staff member, i want to #apply# a #hold#,

   user_story_328: #G03# As a Staff member, i want to #remove# a #hold#,

Original texts of user stories are:
   user_story_327: #G03# As a Staff member, I want to Apply a Hold,
   so that I can prevent progression through the workflow or other
   actions in the system until the issue is resolved.

   user_story_328: #G03# As a Staff member, I want to Remove a Hold,
   so that I can allow progression through the workflow or other actions
   in the system now that the issue has been resolved.
```


Listing 13: Example of generated textual report consist of one conflict pair

Step 3: Evaluation

In this step, we collect all potential conflicts we have found in an Excel file called 'Conflict_Evaluation' to compare the result provided by the tool with the ground truth. See the section 5.6 for a comprehensive explanation.

5.4 Implementation

In this section, we explain the objective and scope of the implementation, the functionality and the programming languages used.

The entire implementation is available in the GitHub repository [39].

Methodology

This section explains and introduces tools that are required during the development process.

Following approach and tools are necessary in order to develop our workflow:

- Java as programming language[40]: Java is a widely used object-oriented programming language.
- GitHub as version control[41]: GitHub is a developer platform that allows developers to create, store, manage and share their code. It uses Git software, providing the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.

Implementation Phases

This section contains a step-by-step guide to implementation, starting with the set-up and ending with the extracting report.

Following steps implies the classes and methods related to org.backlogconflict.code.preparation:

Extracting Parts of USs from the Input Dataset

In this section, we explain the methods we used to convert the US data structured in JSON files into a custom US data structure that splits the elements such as Entity, Action, Contain, Target in terms of their occurrence in the main and benefit part.

Methods of the USPartExtractor Class

In this section, the methods of the USPartExtractor class is described as follows:

- getStringFromOffset: This method was designed to extract a substring from a given main text of US based on the start and end offsets.

  As input it receives the text of US, a start offset and an end offset as an integer and as output it extracts the part of the text of US that starts at "start offset" and ends just before "end offset".

[39] https://github.com/amirrabieyannejad/conflict_analysis_between_USs/tree/main
[40] https://www.java.com/de/
[41] https://github.com/

- runUSPartExtractor: This method processes multiple datasets by reading JSON lines from input files, transforming each JSON object, and then writing the transformed JSON objects to output files.

  As input it receives:

  - An array of dataset names: each name in the array corresponds to a specific dataset that is processed
  - The base directory path in which the dataset files are located. This path is used to construct the full file paths for reading inputs and writing outputs.

  It iterates through each dataset name in the "dataSets" array and constructs the paths of the input and output file based on the dataset name of the "filePath".

  It also reads all lines from the input file and passes them to the `transformJson` method for the transformation process. Finally, it writes the result to the output file.

- transformJson: This method is the main component of the USPartExtractor class. It processes a JSON object that represents a US, extracts and categorises its entities and relationships according to their occurrence in the main or benefit part of the US text and assigns this information in a new JSON structure.

  As input it receive the original JSON object containing information about a US, including text, entities, and relations.

  In this method following steps were carried out:

  - It initialize a JSON object to hold the transformation data and extract the basic information like "Text" and "ID" of US as well as "PID"(project ID)
  - It initialises two JSON objects, namely `Main` and `Benefit`, and specifies the elements and their relationships accordingly.
  - It initialises JSON arrays for various elements such as "Persona", "Entity", "Action" and splits them based on the start and end offsets of the benefit part into "Main" or "Benefit" JSON object.

    If the end offset of an element is smaller than the start offset of the benefit string, it is considered to be an element of the main part, otherwise it is considered to be an element of the benefit part.
  - Iterates through each relation and determines the elements involved. If both elements belong to the main part, a relation is initialised in the "Main" JSON object according to their category (Triggers, Targets, or Contains).

    If both elements belong to the benefit part, a relation is initialised in the "Benefit" JSON object according to their category (Triggers, Targets, or Contains).

    If elements belong to different parts of US, the relation is initialised in a `Mix` JSON object.

  As output a JSON object containing the transformed data will be returned.

Addition of Action Annotations into Dataset

In this section, we explain the classes and their methods that we used to retrieve action annotations associated with the verb from the action annotation database using *VerbFinder* class. Finally, we insert the retrieved action annotations into the JSON structure of the US using *ActionsAnnotationsCreator* class.

Methods of the VerbFinder Class

In this section, the methods of the VerbFinder class is described as follows:

- loadCSV: This method reads a CSV file with verb and action annotation pairs, processes each line and fills a map with these pairs so that when the VerbFinder class is initialised, a map with verb and action annotation pairs is available to retrieve the action annotations as needed.

  It receives a file path of the CSV file to be read as input. The method does not return a value directly.

- getActionAnnotations: This method retrieves the action annotations associated with a specific verb from a pre- populated map (verbMap).

  As input, it receives a verb as a string for which the action annotations are to be retrieved.

  The method then accesses the verbMap and uses the verb provided as a key.

  As output, it receives the corresponding action annotation from the map. If the verb is not available in the verbMap, the method returns `null`.

Methods of the ActionsAnnotationsCreator Class

In this section, the methods of the ActionsAnnotationsCreator class is described as follows:

- addActionsAnnotations: This method reads JSON files for each dataset, initialize the VerbFinder with provided file path, adds action annotations to each JSON object correspond to specific US within these files, and saves the modified JSON objects back to the files.

  As input it receives:

  - An array of dataset names which each name corresponds not only to a subdirectory containing a JSON file to be processed, but also to the file name.
  - The base directory path in which the dataset files are located. This path is used to construct the full file paths for reading inputs and writing outputs.
  - The base directory path where the CSV file is located. This path is used to build the full file paths for reading the database with the action annotations to initialise the map(verbMap).

The method does not return any value. Instead, it modifies the JSON files in place by adding action annotations to each JSON object related to specific US in the file.

- addActionAnnotations: This method processes a JSON object that corresponds to a US in order to add action annotations to the "Targets" and "Contains" relationships within the "Main" part of the US in JSON object.

  As input, it receive a JSON object representing a US with nested structures for "Main" and "Benefit" part, each containing "Targets", "Contains" arrays.

  The method modifies the JSON object by adding the action annotations to the "Targets" and "Contains" relations within the "Main" part.

- findActionsAnnotations: This method retrieves the action annotation associated with a given verb. This method act as a utility for mapping verbs to their corresponding action annotations using the instance VerbFinder class.

  As input, it takes the verb for which the action annotation is to be found. The output returned is a String containing the action annotations for the verb provided.

Figure 37 is a class diagram that illustrates the attributes, operations and relationships related to the org.backlogconflict.code.preparation package.

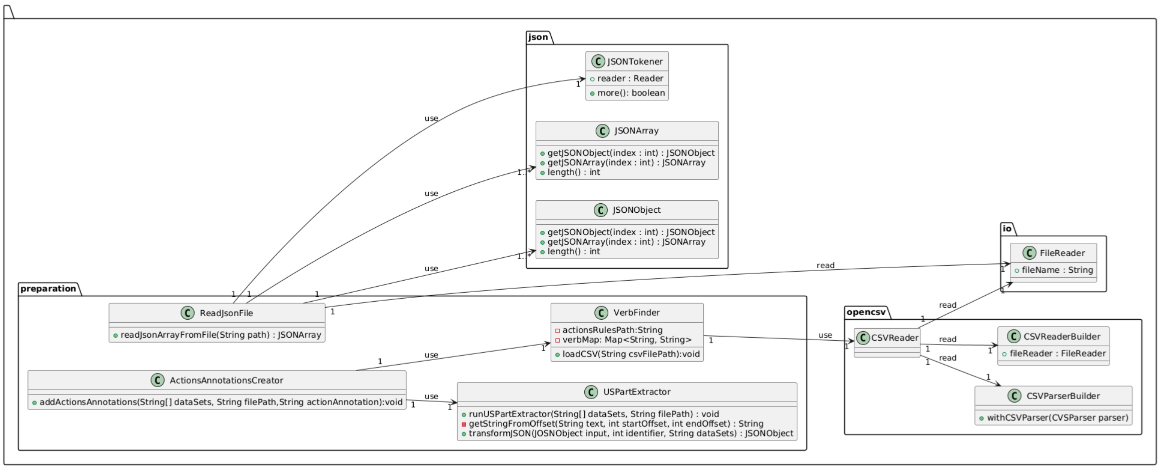


Figure 37: Class diagram of the org.backlogconflict.code.preparation class and its relationships

Storing Conflict Pair into ConflictPair Class

To store the elements of a conflict pair, we implement the class *ConflictPair* to store the information needed to extract a textual report using *ReportMaker* class.

The ConflictPair class contains the following methods that are important for the ReportMaker class:

- get/setActionAnnotationUs1: The setter method stores the *action annotation* for the first US in the conflict pair and the getter method retrieves it.
- get/setActionAnnotationUs2: The setter method stores the *action annotation* for the second US in the conflict pair and the getter method retrieves it.
- get/setActionUs1: The setter method stores the *action* for the first US in the conflict pair and the getter method retrieves it.
- get/setActionUs2: The setter method stores the *action* for the second US in the conflict pair and the getter method retrieves it.
- get/setConflictReason: The setter method stores the *conflict reason* of conflict pair and the getter method retrieves it.
- get/setJsonConflict1: The setter method stores the *JSON object* containing all JSON elements relating to first US, and the getter method retrieves it.
- get/setJsonConflict2: The setter method stores the *JSON object* containing all JSON elements relating to second US, and the getter method retrieves it.
- get/setNounContainUs2: The setter method stores the textitcontain entity of second US, and the getter method retrieves it.
- get/setNounMainUs1: The setter method stores the textitentity of first US, and the getter method retrieves it.
- get/setpId: The setter method stores the *P_ID*(Project ID), and the getter method retrieves it.
- get/setUsId1: The setter method stores the *US_ID* of first US in the conflict pair, and the getter method retrieves it.
- get/setUsId2: The setter method stores the *US_ID* of second US in the conflict pair, and the getter method retrieves it.
- get/setConflictPair1: The setter method stores the *US_Nr* of first US in the conflict pair, and the getter method retrieves it.
- get/setConflictPair2: The setter method stores the *US_Nr* of second US in the conflict pair, and the getter method retrieves it.
- equals(Override): The equals method is overridden to provide a custom definition of equality for ConflictPair objects. Two ConflictPair objects are considered equals if:
    1. They are the same instance.
    2. They are of the same class.

3. Their *conflictPair1* and *conflictPair2* fields are equal.

This custom equality definition is used to compare *ConflictPair* objects based on the value of *conflictPair1* and *conflictPair2*.

- hashCode(Override): This method is overridden to provide a custom hashcode calculation for ConflictPair objects. This hashcode calculation ensures that equal ConflictPair objects (as determined by the *equals* method) have the same hashcode, which ensures that the behaviour of HashMap collections is correct.

ReportMaker Class

To find conflict pairs and store their elements in the ConflictPair object, we must first iterate through each US and verify the US in pairs with the defined criteria.

Figure 38 is a class diagram that illustrates the attributes, operations and relationship related to the extraction of conflict pairs founded by tool and their storage in a ConflictPair object.

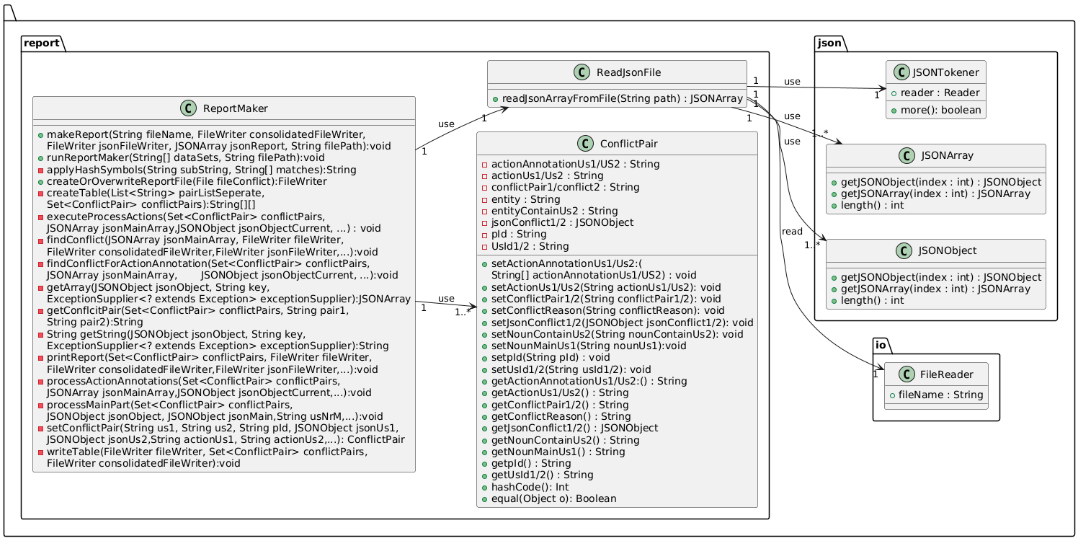


Figure 38: Class diagram related to *org.backlogconflict.code.report* package

Methods related to Extracting and Reporting Conflict Pairs

In order to report conflict pairs such with information such as highlighting USs texts with hash mark(#), conflict pair, affected entity, contain entity related to second US, action of USs, and conflict reason, following methods in the ReprortMaker class are defined:

- applyHashSymbols: This method changes a given substring by surrounding the specified matching elements with hash symbols (#). It ensures that only the words that match the elements in the provided array are changed.

  As input it receive a substring in which the matching elements will be replaced with hash-surrounded versions and an array of string representing the elements to be surrounded by hash symbols within the substring.

  The method returns a modified version of the substring where each specified match is surrounded by hash symbols.

- createOrOverwriteReportFile: This method is designed to create or overwrite a report file. It ensures that a FileWriter object is returned, which can then be used to write to the specified file.

  As input it receive file object representing the file that needs to be created or overwritten.

  As output, it returns a FileWriter object that is set up to write the specified file, regardless of whether it was newly created or overwritten.

- createTable: This method constructs a two-dimensional table (a matrix) that represents conflicts between pairs of USs. The header rows of the table and the first column are filled with modified identifier of USs (e.g. us_xx instead of user_story_xx) from a given list and the cells in the table are filled with `X` based on the conflicts found between pairs of these USs using *getConflcitPair* method.

  As input it receives a list of strings where each string represents a US and a set of ConflictPair objects representing conflicts between pairs of USs.

  As output, the method returns a two-dimensional array that represents the conflict table. The header rows and the first column are identifiers of the conflicting USs and the cells contain `X` if there is a conflict between these two USs.

- getConflcitPair: This method searches for conflicts between a pair of US-pairs and returns `X` if there is a conflict between two USs, otherwise it returns a white space.

- runReportMaker: The runReportMaker method generates conflict reports for a number of datasets using the *makeReport* method. It creates both a consolidated text report and a JSON report that summarises the conflicts found in the specified datasets.

  As input, it receives an array with the names of the datasets to be processed and the path of the base file in which the datasets are located.

  As output, it produces a text file containing the consolidated conflict report and a JSON file containing the conflict report in JSON format.

- makeReport: This method generates a conflict report for a single dataset. It reads the dataset from input JSON file, processes it to find conflicts using *findConflict* method, and writes the results to both a text file and a JSON report.

  It receives the following inputs:

  - fileName: The name of the dataset file to process.
  - consolidatedFileWriter: A FileWriter object to write the consolidated conflict report for all datasets.
  - jsonFileWriter: A FileWriter object to write the JSON conflict report for single dataset.
  - jsonReport: A JSON array object for collecting conflict reports from a dataset.
  - filePath: The base file path where the dataset files are located.

  As output, the method updates the consolidated conflict report file and the JSON conflict report with the conflicts found in the specified dataset.

  A separate conflict report text file for the individual dataset is created or overwritten with the conflict details.

- findConflict: This method analyses a given JSON array to identify and report conflicts within the USs. It iterates through each JSON object which represents the US in the array, processes the "Main" part of each JSON object, and collects conflict information using *processMainPart* method. The results are then written to specified file writers and a JSON report using *printReport* method.

  As input, the same input related to the makeReport method is passed.

  As output, The method populates the provided FileWriter objects and JSONArray with conflict information derived from the input JSON array. It identifies conflicts by processing the "Main" part of each JSON object and aggregates the results for reporting.

- processMainPart: This method processes the "Main" part of a given JSON object, extracting various arrays and strings using *getArray* as well as getString methods. It then checks for action annotations using *processActionAnnotations* method and processes them to identify conflicts. The method adds the conflicts to a provided set of conflict pairs.

  It receives the following inputs:

  - conflictPairs: A set to store the identified conflict pairs.
  - jsonObject: The JSON object currently being processed representing specified US.
  - jsonMain: The "Main" part of the JSON object(related to the specified US) currently being processed.
  - usNrM: The US identifier of the current JSON object.

- jsonMainArray: The main JSON array containing all the USs to be analysed.
- conflicts: A set US identifier to store unique conflict pair.

The method does not return a value.

- processActionAnnotations: This method processes action annotations from JSON objects that refer to USs in order to process action annotations. Iterates through the element *Target Action Annotations*, checks whether there are multiple action annotations and executes the associated actions to process these annotations. If an action annotation that refers to an action contains multiple actions, the *executeProcessActions* method is called for each part of the annotation. Identified conflicts are added to a provided set of conflict pairs.

  It receives the following inputs:

  - usNrM: The US identifier of the current JSON object.
  - jsonObject: The JSON object currently being processed which corresponds to specified US.
  - actionArray: The array of actions in the US JSON object.
  - entityArray: The array of entities in the US JSON object.
  - triggersArray: The array of triggers in the US JSON object.
  - targetsArray: The array of targets in the US JSON object.
  - containsArray: The array of contains in the US JSON object.
  - text: The text associated with the US JSON object.
  - persona: The array of personas in the US JSON object.
  - conflicts: A set to store unique conflict strings.

  The method does not return a value.

- executeProcessActions: This method processes specific action annotations (like preserve, delete, create, and forbid) by checking for conflicts with other annotations in the JSON data. It uses the *findConflictForActionAnnotation* method to identify and handle these conflicts, adding them to a set of conflict pairs.

  The method uses a switch statement to process the action annotation based on the defined criteria and handle it accordingly. For each case in the switch statement, the method calls *findConflictForActionAnnotation* with different parameters to identify potential conflicts based on specific criteria.

  It receives the following inputs:

  - usNrM: The US identifier of the current JSON object.
  - jsonObject: The JSON object currently being processed which corresponds to specified US.

- actionArray: The array of actions in the US JSON object.
- entityArray: The array of entities in the US JSON object.
- triggersArray: The array of triggers in the US JSON object.
- targetsArray: The array of targets in the US JSON object.
- containsArray: The array of contains in the US JSON object.
- text: The text associated with the US JSON object.
- persona: The array of personas in the US JSON object.
- conflicts: A set US identifier to store unique conflict pair.
- verb: The verb extracted from the *Target Action Annotations*.
- noun: The noun extracted from the *Target Action Annotations*.
- actionAnnotation: The action annotation (e.g., "preserve", "delete").
- nounMainUs1: The entity of first US.

The method does not return a value.

• findConflictForActionAnnotation: This method is responsible for identifying conflicts between USs based on certain action annotations and entity relationships within a JSON dataset. Iterates through all JSON objects in a dataset to compare the current US with others. It iterates through "Target Action Annotations" to compare action annotations and entities. It also iterates through the "Contain Action Annotations" array to compare the contained entities. If a conflict is detected between two US, the *setConflictPair* method is used to create a ConflictPair object with specific information and add the conflictPair object to a set called *conflictPairs*.

  It receives the following inputs:

  - conflictPairs: A set to store identified conflict pairs.
  - jsonMainArray: The main JSON array containing all the USs to be analysed.
  - jsonObjectCurrent: The current US JSON object being processed.
  - actionAnnotationUs1: The action annotation from the first US.
  - criticalActionAnnotation: The critical action annotation that triggers conflicts.
  - entityUs1: The entity associated with the first US.
  - actionUs1: The action associated with the first US.
  - conflicts: A set US identifier to store unique conflict pair.

  The method does not return a value.

- setConflictPair: This method effectively initializes and populates a *ConflictPair* object with the provided parameters, encapsulating all necessary information about a conflict pair between two USs.

  It receives the following inputs:

  - us1, us2: Identifiers for the conflicting USs.
  - pId: Project ID associated with the USs.
  - jsonUs1, jsonUs2: JSON objects representing the conflicting USs.
  - actionUs1, actionUs2: Actions associated with the conflicting USs.
  - actionAnnotationUs1, actionAnnotationUs2: Action annotations related to the conflicting actions.
  - conflictReason: Reason of the conflict.
  - nounMainUs1: Noun associated with the first US.
  - nounContainUs2: Noun contained within the second US, if applicable.
  - usId1, usId2: User story IDs associated with first and second USs, respectively.

  As output, it returns the populated ConflictPair object to the caller.

- getArray: This method is designed to retrieve a JSON array from a JSON object based on a specified key.

  It receives the following inputs:

  - jsonObject: The JSON object to retrieve the JSON array.
  - key: The key whose associated JSON array is to be retrieved from JSON object.
  - exceptionSupplier: A functional interface that can supply an exception of a specific type if needed. This is used to handle cases where the expected key is not found in the JSON object.

  If the key exists in the JSON object, the corresponding JSON array is retrieved as output, otherwise an exception is thrown.

- getString: This method is designed to retrieve a String from a JSON object based on a specified key.

  It receives the following inputs:

  - jsonObject: The JSON object to retrieve the string.
  - key: The key whose associated string is to be retrieved from JSON object.
  - exceptionSupplier: A functional interface that can supply an exception of a specific type if needed. This is used to handle cases where the expected key is not found in the JSON object.

If the key exists in the JSON object, the corresponding string is retrieved as output, otherwise an exception is thrown.

- printReport: This method is responsible for creating a textual, tabular and JSON report based on the conflict pairs found during processing. For each ConflictPair, it extracts detailed information such as IDs, text of the USs, actions, annotations, conflict reasons, highlighting of the affected elements in the texts of the USs in the main parts using the *applyHashSymbols* method.

  It receive following inputs:

  - conflictPairs: A set of ConflictPair objects.
  - fileWriter, consolidatedFileWriter, jsonFileWriter: These are FileWriter objects responsible for writing text and JSON data to respective files.
  - fileName: A String representing the name of the dataset being processed.

  The method does not return a value.

Import Result for Evaluation

In this step, we collect all potential conflicts found in an Excel file called 'Conflict_Evaluation' by creating a report in JSON format and using the following VBA script[42] to import the required information into the Excel file: Listing 14 illustrates the implemented VBA script that uses the JSON report to import the required information into an Excel file.

[42] https://github.com/amirrabieyannejad/conflict_analysis_between_USs/blob/main/00_annotated_datasets/extractFromJSONFiles_Version(new).xlsm

```
Sub ParseJSON()
    Dim FSO As Object
    Dim JsonTS As Object
    Dim JsonText As String
    Dim JsonObj As Object
    Dim i As Integer
    Dim j As Integer
    Dim lastRow As Long
    Dim ArrayDataSets() As Variant

    'Initialize FileSystemObject
    Set FSO = CreateObject("Scripting.FileSystemObject")
   'Assign the absolute path to the JSON file consolidated_conflict_report
    Set JsonTS = FSO.OpenTextFile(
    "C:\...\org.backlogconflict\00_annotated_datasets\consolidated_conflict_report
    .json", 1)
    JsonText = JsonTS.ReadAll
    JsonTS.Close
    'Parsing the JSON file using JSONConverter
    Set JsonObj = JsonConverter.ParseJSON(JsonText)
    'Read the JSON file and insert the required information into an Excel file
    With ThisWorkbook.Sheets("Sheet2")
     lastRow = .Cells(.Rows.Count, "A").End(xlUp).Row
     For j = 1 To JsonObj.Count
      .Cells(lastRow + j, 1).Value = JsonObj(j)("PID")
      .Cells(lastRow + j, 2).Value = JsonObj(j)("Conflict Pair")
      .Cells(lastRow + j, 3).Value = JsonObj(j)("Texts of Main Parts")("US1") &
      vbCrLf & vbCrLf & JsonObj(j)("Texts of Main Parts")("US2")
      .Cells(lastRow + j, 4).Value = JsonObj(j)("Conflict-Reason")
      .Cells(lastRow + j, 5).Value = JsonObj(j)("Entity of US1")
      .Cells(lastRow + j, 6).Value = JsonObj(j)("Contain US2 Entity")
      .Cells(lastRow + j, 7).Value = JsonObj(j)("Actions")("Action US1")
      .Cells(lastRow + j, 8).Value = JsonObj(j)("Action Annotations")("US1")
      .Cells(lastRow + j, 9).Value = JsonObj(j)("Actions")("Action US2")
      .Cells(lastRow + j, 10).Value = JsonObj(j)("Action Annotations")("US2")
      .Cells(lastRow + j, 11).Value = JsonObj(j)("Conflict Pair Old")


        Next j
      End With
    ' Clean up
    Set FSO = Nothing
    Set JsonTS = Nothing
    Set JsonObj = Nothing
End Sub
```

Listing 14: VBA script for importing information from a JSON report into an Excel file

Error Handling

There were erroneous data in the datasets forcing us to handle them correctly. Therefore, we implement/use the following exceptions to accurately distinguish and handle them.

The following classes, which extend the *Exception* class, are used in *org.backlogconflict. code.preparation* and *org.backlogconflict. code.report* packages:

- MainPartInJsonFileNotFound: Is triggered if the JSON object "Main" could not be found in the US JSON object.

- ActionAnnotationInJsonFileNotFound: Is triggered if the entry *Action Annotations*, which contains *Targets/Contains Action Annotations*, is not present in the US JSON object and its absence should be reported.
- ActionInJsonFileNotFound: Is triggered if the entry *Action* is not present in the US JSON object and its absence should be reported.
- ContainsInJsonFileNotFound: Is triggered if the entry in *Contains* is not present in the US JSON object and its absence should be reported.
- EntityInJsonFileNotFound: Is triggered if the entry *Entity* is not present in the US JSON object and its absence should be reported.
- PersonaInJsonFileNotFound: Is triggered if the entry *Persona* does not exist in the US JSON object for a specified US.
- TextInJsonFileNotFound: Is triggered if the entry *Text* does not exist in the US JSON object for a specified US.
- UsNrInJsonFileNotFound: Is triggered if the entry *Us_Nr* does not exist in the US JSON object for a specified US.
- TargetsInJsonFileNotFound: Is triggered if the entry *Targets* does not exist in the US JSON object for a specified US.
- TriggersInJsonFileNotFound: Is triggered if the entry *Triggers* does not exist in the US JSON object for a specific US.
- EmptyOrNotExistJsonFile: Is triggered if the JSON file is empty or does not exist.
- EntityInRelationsNotFound: Is triggered if the entity as part of *relation* JSON array is not present in the US JSON object and its absence should be reported.

5.5 Test

Our objectives in this section include validating certain functions, checking the system requirements and ensuring the reliability and robustness of the implemented classes and methods.

As part of our testing strategy, we perform unit tests with *JUnit* version 4[43], which we selected for its compatibility with Eclipse version 2023-03.

EclEmma[44] We have integrated a code coverage tool into the Eclipse IDE to ensure thorough testing and coverage. With the help of EclEmma, we systematically measured

[43] https://junit.org/junit4/
[44] https://www.eclemma.org/

the effectiveness of our test suites and determined the test coverage for each individual class.

Configuration of Test Environment

In the main project *org.backlog*, we create a separate package called *org.backlogconflict.test*. This package contains the following Java classes, which correspond to the respective Java source code files:

- USPartExtractorTest
- ActionsAnnotationsCreatorTest
- VerbFinderTest
- ReportMakerTest

Scope of Testing

The scope of the tests depends on the system requirements and the implemented classes and their methods. The implemented error handling classes are also tested.

Test Cases and their Code Coverage

We describe the individual test cases that are carried out during the test process. Each test case contains a description of the test scenario, the data provided and the expected result. To refine and improve our test cases, we used a code coverage report created by EclEmma to increase coverage and ensure a more reliable, error-resistant application.

Table 20 shows the test cases for the USPartExtractor.java class and Figure 39 shows the code coverage.

Test Case	Supplied Data	Expected Outcome	Description
BenefitNotExist	Assign USs without benefit part	The length of the entries(action, entity, contains, targets) for the benefit should be null	Check what happens if the US has no benefit part
EntityInRelationNotFound (Case1)	Preparing a JSON object whose entity entry is not defined as an entity in the relation entries assigned as the first reference	Through an exception: *NullPointerException*	Checks whether the first referencing entity is already defined as an entity in the relation entries
EntityInRelationNotFound (Case2)	Preparing a JSON object whose entity entry is not defined as an entity in the relation entries assigned as the second reference	Through an exception: *NullPointerException*	Checks whether the second referencing entity is already defined as an entity in the relation entries

Test Case	Supplied Data	Expected Outcome	Description
EmptyOrNotExistJsonFile	Assign an empty JSON file	Through an exception: *NoSuchFileException*	Checks whether the JSON file is empty
testPid	JSON file with the project ID	Create JSON object "PID" which contain the project ID of US	Checks whether the project ID was created as expected
UnexpectedType	JSON file with a US with unexpected type	Through an exception: *JSONException*	Checks whether the expected types (action, entity, text, etc.) in the corresponding US pass on in the JSON file and not unexpected

Table 20: Test cases for USPartExtractor class

JSONTransformer

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
JSONTransformer()		0 %		n/a	1	1	1	1	1	1
transformJson(JSONObject, int, String)		100 %		92 %	5	39	0	121	0	1
runJsonTransformer(String[], String)		100 %		100 %	0	3	0	14	0	1
getStringFromOffset(String, int, int)		100 %		n/a	0	1	0	1	0	1
Total	3 of 629	99 %	5 of 74	93 %	6	44	1	137	1	4

Figure 39: Code coverage related to class USPartExtractor

Table 21 shows the test cases for the VerbFinder.java class and Figure 40 shows the code coverage.

Test Case	Supplied Data	Expected Outcome	Description
testLoadCSV (Case1)	Assign a CSV file that contains two columns: one for verbs and one for action comments, so that both columns contain no empty cells	Action Annotation corresponds to verbs should be returned	Checks whether the action note has been returned accordingly in respect of the verb
testLoadCSV (Case2)	Assign a CSV file that contains two columns: one for verbs and one for action comments, so that one column contain empty cell	A verb without an assigned action annotation should be ignored	Checks whether the action annotation has been returned accordingly in respect of the verb

Table 21: Test cases for VerbFinder class

Table 22 shows the test cases for the ActionsAnnotationsCreator.java class and Figure 41 shows the code coverage.

VerbFinder

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
loadCSV(String)		100 %		100 %	0	3	0	8	0	1
VerbFinder(String)		100 %		n/a	0	1	0	4	0	1
getActionAnnotations(String)		100 %		n/a	0	1	0	1	0	1
Total	0 of 63	100 %	0 of 4	100 %	0	5	0	13	0	3

Figure 40: Code coverage related to class VerbFinder

Test Case	Supplied Data	Expected Outcome	Description
ActionsAnnotationsCreator	Assign a JSON file without entries of action annotations (target or contains action annotation)	Create action annotation related to target action annotated	Checks whether the JSON array for the corresponding action has been created for the target

Table 22: Test cases for ActionsAnnotationsCreator class

ActionsAnnotationsCreator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
addActionAnnotations(JSONObject)		100 %		100 %	0	9	0	51	0	1
addActionsAnnotations(String[], String, String)		100 %		100 %	0	3	0	11	0	1
findActionsAnnotations(String)		100 %		n/a	0	1	0	1	0	1
ActionsAnnotationsCreator()		100 %		n/a	0	1	0	2	0	1
Total	0 of 311	100 %	0 of 20	100 %	0	14	0	65	0	4

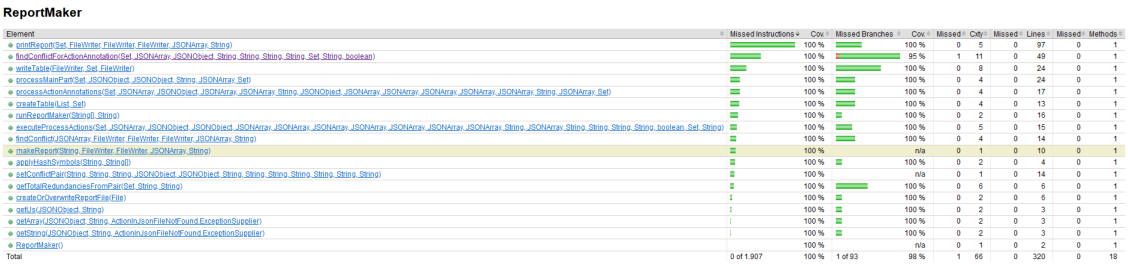
Figure 41: Code coverage related to class ActionsAnnotationsCreator

Table 23 shows the test cases for the Evaluation.java class and Figure 42 shows the code coverage.

Test Case	Supplied Data	Expected Outcome	Description
ActionAnnotation InJsonFileNotFound	Provision of a JSON object without action annotation	Through an exception: *ActionAnnotation InJsonFileNotFound*	Checks whether the JSON object provided contains an entry for action annotations
TargetActionAnnotation InJsonFileNotFound	Provision of a JSON object without target action annotation	Through an exception: *ActionAnnotation InJsonFileNotFound*	Checks whether the JSON object provided has an entry for target action annotations
ContainActionAnnotation InJsonFileNotFound	Provision of a JSON object without contain action annotation	Through an exception: *ActionAnnotation InJsonFileNotFound*	Checks whether the JSON object provided has an entry for contain action annotations
EntityContainExist	Provision of a conflict pair with indirect conflict through contain relation	The US-pairs should be reported as a conflict pair	Checks whether the conflict is recognised if the entity from Us belongs to contain relation
JsonArrayNotFound	Provision of a JSON file in which a JSON array is missing	Null should be return	Checks whether the specific JSON array was not found, if so, return null
JsonObjectNotFound	Provision of a JSON file in which a JSON object is missing	Null should be return	Checks whether the specific JSON object was not found, if so, return null
runReportMaker_Main	Provision of two USs that contradict each other	The conflict pair should be reported in text form	Checks whether two USs that contradict each other have already been reported as conflict pair

Test Case	Supplied Data	Expected Outcome	Description
MainIsEmpty	Provision of a JSON object with empty main part entry	Through an exception: *MainPartInJsonFile NotFound*	Checks whether the main part entry in provided JSON object is not empty
MainNotExist	Provision of a JSON object without main part entry	Through an exception: *MainPartInJsonFile NotFound*	Checks whether the JSON object provided has main part entry
runReportMaker_NoConflict	Provision of a set of USs that not contradict each other	The conflict pair should not be reported	Checks whether two USs that do not contradict each other should not be reported as a conflict pair
UsNrInJsonFileNotFound	Provision of a JSON object that don't have US identifier	Through an exception: *UsNrInJsonFileNotFound*	Checks whether the JSON object in the JSON file already has an identifier
writeTable	Provision of a conflict pair that should be reported	Conflict pair should be listed in tabular form	Check whether the conflict pair is already listed in the table

Table 23: Test cases for ReportMaker class


ReportMaker

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
printReport(Set, FileWriter, FileWriter, FileWriter, JSONArray, String)		100 %		100 %	0	5	0	97	0	1
findConflictForActionAnnotation(Set, JSONArray, JSONObject, String, String, String, String, Set, String, boolean)		100 %		95 %	1	11	0	49	0	1
writeTable(FileWriter, Set, FileWriter)		100 %		100 %	0	8	0	24	0	1
processMainPart(Set, JSONObject, JSONObject, String, JSONArray, Set)		100 %		100 %	0	4	0	24	0	1
processActionAnnotations(Set, JSONArray, JSONObject, JSONArray, JSONArray, String, JSONObject, JSONArray, JSONArray, JSONArray, JSONArray, JSONArray, String, JSONArray, Set)		100 %		100 %	0	4	0	17	0	1
createTable(List, Set)		100 %		100 %	0	4	0	13	0	1
runReportMaker(String[], String)		100 %		100 %	0	2	0	16	0	1
executeProcessActions(Set, JSONArray, JSONObject, JSONObject, JSONArray, JSONArray, JSONArray, JSONArray, JSONArray, JSONArray, JSONArray, String, JSONArray, String, String, String, String, boolean, Set, String)		100 %		100 %	0	5	0	15	0	1
findConflict(JSONArray, FileWriter, FileWriter, FileWriter, JSONArray, String)		100 %		100 %	0	4	0	14	0	1
makeReport(String, FileWriter, FileWriter, JSONArray, String)		100 %		n/a	0	1	0	10	0	1
applyHashSymbols(String, String[])		100 %		100 %	0	2	0	4	0	1
setConflictPair(String, String, String, JSONObject, JSONObject, String, String, String, String, String, String, String)		100 %		n/a	0	1	0	14	0	1
getTotalRedundanciesFromPair(Set, String, String)		100 %		100 %	0	6	0	6	0	1
createOrOverwriteReportFile(File)		100 %		100 %	0	2	0	6	0	1
getUs(JSONObject, String)		100 %		100 %	0	2	0	3	0	1
getArray(JSONObject, String, ActionInJsonFileNotFound.ExceptionSupplier)		100 %		100 %	0	2	0	3	0	1
getString(JSONObject, String, ActionInJsonFileNotFound.ExceptionSupplier)		100 %		100 %	0	2	0	3	0	1
ReportMaker()		100 %		n/a	0	1	0	2	0	1
Total	0 of 1.907	100 %	1 of 93	98 %	1	66	0	320	0	18

Figure 42: Code coverage related to class ReportMaker

5.6 Evaluation

In this section, we address two research questions and explain how we designed and conducted the experiments for each question. We then analyse the results to answer these questions, discuss their significance and share the findings from the results.

Research Questions

The research questions addressed in this section are as follows:

1. Does our tool consistently identify conflicts between USs?

2. How does the performance of the tool change as the count of USs in a backlog increases?

Methodology

To answer the first question, 'Does our tool consistently detect conflicts between USs?', we recapitulate the methodology used to analyse conflicts between USs. We used a systematic approach that includes several important steps:

- Data Collection: For a comprehensive assessment, we applied our approach to 19 backlog datasets presented by Mosser et al. [45]. They applied the Doccano approach to these publicly available requirements datasets[12].

  It is also worth noting that some backlog datasets (g02, g13, g17, g27) did not follow the expected sentence structure, so we did not include them in the evaluation results to avoid unexpected behaviour. Table 24 shows the project number of each dataset and the count of USs.

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
Project Nr.	g03	g04	g05	g08	g10	g11	g12	g14	g16	g18	g19	g21	g22	g23	g24	g25	g26	g27	g28	Total USs
Total USs	57	51	53	66	97	73	54	67	66	102	137	69	83	56	53	100	100	114	60	1458

Table 24: Project number and count of USs contained in each backlog dataset

- Splitting the elements into main and benefit parts: each element of the US (action, entity and relationships) was split into a main part and a benefit part, where the main part represents the core functionality and the benefit part describes the value for the persona. In this analysis, we only consider the conflict between the main parts of the USs.

- Recognition of conflicts: Detection of conflicts between USs in main parts of the USs based on defined criteria.

Ground Truth

To answer the first research question, our automated system needs a reference point to measure its accuracy. This reference point, called 'ground truth', comes from a personal judgement and serves as a benchmark for evaluating the tool's performance.

Unlike the tool evaluation, we did not assess conflict based on defined criteria. Instead, in the ground truth evaluation, we examined conflict based on the text of the US-pairs detected by tool. This involved reading the text of the USs and evaluating conflicts between US-pairs in a semantic manner.

The ground truth serves as the final assessment against which the automated results are compared. It results from the personal judgement of a subject matter expert (in this case me) using a combination of expertise and experience. This personal judgement

[45] https://github.com/ace-design/nlp-stories

is intended to provide a reliable and accurate reference for identifying conflicts between USs.

Table 25 shows the summary of US conflict pairs as ground truth.

Project Nr.	G03	G04	G05	G08	G10	G11	G12	G14	G16	G18	G19	G21	G22	G23	G24	G25	G26	G27	G28	Grand Total
Ground Truth	1	0	3	4	4	1	7	22	26	8	3	1	10	3	10	7	9	24	4	**143**

Table 25: Details of the count of correctly recognised and validated conflict pairs by the tool as ground truth

Evaluate conflict Analysis Delivered by the Tool Through Ground Truth

The automated tool was developed to recognise conflicts between USs based on predefined criteria. It analyses the text and structure of USs to identify discrepancies that could indicate a conflict.

In contrast to ground truth, which is based on the semantic comparison of texts in the main parts of USs, the evaluation of the tool relies heavily on specified labelling (targets, triggers, contains) using the Doccano tool and predefined criteria.

Table 16 shows the aggregation of the US conflict pairs found in the main parts evaluated by the tool.

Project Nr.		G03	G04	G05	G08	G10	G11	G12	G14	G16	G18	G19	G21	G22	G23	G24	G25	G26	G27	G28	Grand Total
Conflict Pairs Detected by Tool	Overall	1	0	3	4	4	1	8	22	27	16	5	1	19	4	14	8	9	27	5	**173**

Table 26: Aggregation of the count of US conflict pairs in relation to the main parts assessed by the tool

Assessment of Result: High-Level Overview When comparing the results provided by the tools with the ground truth, we found that 143 US conflict pairs were correctly assessed as US conflict pairs and 30 US conflict pairs were invalid on the basis of the ground truth.

Table 27 shows the count of valid and invalid US conflict pairs recognised by the tool and assessed by the ground truth.

Based on the datasets provided, 173 US conflict pairs were found across all projects, with the highest count found in the backlog G27 dataset(24 conflict pairs), indicating a significant occurrence of conflict in the USs of this project.

Assessment of Result: Detailed Insights In the USs of the datasets, there are some verbs such as 'have', 'know' and nouns such as 'data', 'resource', 'collection', 'list',

Project Nr.		G03	G04	G05	G08	G10	G11	G12	G14	G16	G18	G19	G21	G22	G23	G24	G25	G26	G27	G28	Sub/Grand Total	
Conflict Pairs Detected by Tool	Overall	1	0	3	4	4	1	8	22	27	16	5	1	19	4	14	8	9	27	5	**173**	
Ground Truth	Valid Conflict Pairs	1	0	3	4	4	1	7	22	26	8	3	1	10	3	10	7	9	24	4	**143**	**173**
	Unvalid Conflict Pairs	0	0	0	0	0	0	1	0	1	8	2	0	9	1	4	1	0	3	1	**30**	

Table 27: Detail on the count of discrepancies between the result provided by the tool and the ground truth

and 'file' that are cross-contextual, so they do not contain generic information, which has a very negative impact on conflict detection (30 recognised US conflict pairs are false positives).

Example 5.4. *For example, user_story_983 and user_story_989 are reported as a conflict pair in the G24 backlog dataset:*
user_story_1365: #G24# As a depositor, I want to store and manage #datasets# via a simple web interface.
user_story_1409: #G24# As a research information manager, I want to have #datasets# linked to metadata about projects.
The depositor needs a simple interface for uploading and managing datasets. The research information manager needs to be able to link these datasets to comprehensive metadata about the projects.
These needs are not inherently conflicting, but they require careful design consideration to ensure that both needs are met effectively.
To avoid the conflict, the verb "have" related to "user_story_1409" should be defined more precisely. The US should be amended as follows:
user_story_1409: #G24# As Research Information Manager, I want to `link datasets` *to metadata about projects.*

There are also annotated USs with inaccurate labelling by the Doccano tool. In other words, there is more information about the resource in concern provided in US, which leads to inconsistencies and bias.

Example 5.5. *In the dadaset of backlog G25, for example, we have a US-pair that are marked as conflict:*
user_story_1506: "#G25# As a DAMS manager, I want to know when the #application# of a statute to an object or object component has been #modified#, either manually or automatically."
user_story_1510: "#G25# As a DAMS manager, I want to know if #application# of a library policy to an object or object component has been #modified#, either manually or automatically. "

In the annotated dataset, the identified entities of "user_story_1506" and "user_story_1510" are only 'Application', which is not fully labelled. The correct labelling for "user_story_1506" should be `"Application of a statute"` *and for "user_story_1510" should be* `"Application of a library policy"`. *With these changes, there will be no conflict between these two USs.*

Research question 1: Conclusion After comparing the ground truth result with the result provided by our tool, we find that 83% of the conflict pairs found are valid and only 17% of the conflict pairs are invalid.

The results of the automated tool are consistent with the ground truth, which shows that it reliably recognises conflicts. This agreement with personal judgements increases confidence in the accuracy and validity of the tool.

The tool also demonstrates its effectiveness and trustworthiness. This reliability means that users can rely on the tool to accurately identify conflicts, reducing the need for extensive manual review.

Performance Evaluation

To answer the second research question "How does the performance of the tool change as the count of USs in a backlog increases?", we conducted a series of tests to measure the time it takes the tool to process different counts of USs. This section describes the test method, the results obtained and the impact on the scalability of the tool.

Test Methodology To evaluate the tool's performance, we conducted a set of experiments in which the tool processed different numbers of USs in a backlog. The tests involved the following steps:

1. Backlog Setup: We used backlogs with varying numbers of USs around—50, 70, 90, 120 and 140—to simulate different workload sizes. Each backlog contained USs with varying content, and complexity to represent a realistic range of cases. Table 28 shows information on the backlog data records provided for the performance test application.

US Information	Project Nr.	G04	G11	G10	G27	G19	**Grand Total**
	Number of US	51	73	97	114	137	**472**
	Number of Conflicts	0	1	4	27	5	**37**

Table 28: Information on the backlog datasets provided for the application of the performance test

2. Tool Execution: Each part of toolchain(USPartExtractor, ActionsAnnotationsCreator, and ReportMaker) was run for each backlog and the total time taken to process the entire backlog was recorded. The performance of the tool was measured

by the processing time, i.e. the total time taken to process all USs in the backlog and identify conflicts and reporting.

3. Repeating Tests: To ensure reliability, each test was conducted multiple times, and the average processing time was calculated.

The test environment consisted of:

- Processor: Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz (8 CPUs), 2.0GHz
- Memory: 8070MB RAM
- Display Devices: Intel(R) UHD Graphics 620, 4163 MB(Display Memory)
- Hard Disk: INTEL SSDPEKNW512G8H
- Operating System: Windows 11 Home 64-bit (10.0, Build 22631) (22621.ni_release.220506-1250)
- System Type: 64-bit operating system, x64-based processor

Table 29 shows the result of tool's performance in seconds which were conducted on a controlled environment to ensure consistency.

Test Result	Project Nr.	G04	G11	G10	G27	G19	**Grand Total**
	USPartExtractor	0,2	0,07	0,08	0,09	0,06	**0,5**
	ActionsAnnotations Creator	0,04	0,03	0,04	0,05	0,04	**0,2**
	ReportMaker	0,05	0,05	0,06	0,17	0,08	**0,41**
	Grand Total	**0,29**	**0,15**	**0,18**	**0,31**	**0,18**	**1,11**

Table 29: Information about the result of the tool's performance test, which was measured using the processing time in seconds.

Research question 2: Conclusion There is no direct relationship between the count of USs in a backlog and the processing time required by our tool(G27 vs G19). However, there is a direct relationship between the count of conflicts found between USs and the processing time. The more conflicts there are, the longer the processing time.

This means that developers and project managers with a growing backlog can expect the tool to take a reasonable amount of time to assess conflicts.

Threats to Validity

Several potential threats to validity need to be considered when assessing the conflict between USs both by personal judgement (ground truth) and by the automated tool. This section outlines the main threats to validity and describes how they were mitigated during the study.

Construct validity It refers to how accurately the assessment measures what it is supposed to measure. The following risks have been identified:

- Ambiguity of criteria: If the criteria for conflict are unclear or open to interpretation, this can lead to inconsistent scores. To avoid this, we defined clear and detailed criteria for identifying conflicts in USs.
- Ambiguity of action annotations: If the action annotations associated with each verb in a US are unclear or not specific to the context, this can lead to inconsistent scores. To prevent this, we assign the action annotations based on the context of the backlog instead of using general terms.
- Subjectivity in ground truth: Since the ground truth is based on personal judgements, subjectivity could lead to bias. To minimise this risk, the evaluator (in this case me) has cross-checked the evaluations several times.

External validity External validity refers to how well the results of the study can be transferred to other contexts or populations. Threats to external validity include:

- specificity of USs: If the USs in the study are too specific or specific to the context, the results may not be transferable to other projects. To avoid this, we analysed 19 backlog datasets with different USs and project types.
- Tool Limitation
  - The automated tool is tailored to a specific format of USs, which limits its wider applicability. It is highly dependent on the exact structure of USs (e.g. 'As <user>I want to ... so that ...'), which is crucial for its functionality. Therefore, we evaluated the tool in controlled environments, focussing exclusively on well-structured USs, and investigated its performance in different scenarios.
  - Our tool relies on a specific type of annotations for USs, e.g. action, entity, their reference targets, triggers and contains. The effectiveness of the tool depends on these annotations being accurate and consistent. If the annotations are incomplete or incorrect, the tool may not work properly. Also, the tool may not be compatible with other annotation schemes that use different labels.

  - The verbs in the action annotation database are not included in their root form, but as they actually occur in the USs. This means that the same verb in different forms may not be found in the database. It is also worth noting that some verbs may be missing from the database.

Internal Validity Internal validity is concerned with whether the observed results are attributable to the factors analysed or are influenced by other variables. Potential threats to internal validity include:

- Confounding factors: External factors or unintended variables can influence the evaluation of the conflict analysis. In particular, the USs annotated with Doccano have a significant influence on the conflict analysis. The more phrases are covered as tables (especially as entity and action), the better the evaluation. Since our tool uses Doccano-annotated USs without changes as primary input, these discrepancies are unavoidable.

- Limitation of tool: The automated tool does not take into account the analysis of conflicts in the benefit parts of the USs.

5.7 Conclusion

In this study, we developed and deployed a method that integrates the Doccano tool and our custom tool to systematically identify and report conflicts between USs in software development projects. We also conducted an evaluation of this approach.

By carefully analysing 19 different backlog datasets, our method not only separated the USs into main and benefit parts for nuanced examination, but also facilitated conflict analysis by translating the verbs of the main parts into four distinct categories, namely 'delete', 'create', 'forbid' and 'preserve', and finally reported the potential conflict in text base format.

Our results reveal a decisive finding: the effectiveness of conflict detection is significantly influenced by the quality of the USs and their annotations. Well-formulated USs, in which general verbs (such as 'have', 'know') and nouns (such as 'data', 'list', 'collection') are avoided, as well as a concisely annotated backlog with precise labelling of the entities (nouns) significantly improve the effectiveness of the conflict analysis.

If the main parts of a US pair are contradict each other, then applying one US cause negative effect on another US with deleting a resource which another US are using, or creating a resource which another US prohibits, or delete a resource which another US also wants to delete.

If conflicts between USs are identified, this is a signal for the project team to take a closer look at the requirements and the design of the system. By prioritising, sequencing, defining clear rules, implementing conflict resolution mechanisms, refining USs and involving stakeholders, the team can effectively manage and resolve these conflicts to ensure a well-functioning system.

In summary, our study confirms the central role of a semantic analysis approach in the detection and management of conflicts in project backlogs and thus contributes to the rationalisation of software development processes.

The quality of the annotated USs and their well-formed structure are central to the success of this method. The results of this study provide useful guidance for improving current practices and shaping future research in software project management.

6 Conclusion

In this thesis, we addressed the challenges of managing redundancies and conflicts in USs within agile software development. Our work focused on developing two comprehensive workflows that aim to increase the efficiency of US management through the use of advanced tools and methods specially graph-based annotation of USs.

First, we proposed a redundancy detection framework that combines the Doccano tool, Henshin, and the CDA tool. By analysing 19 different backlog datasets, we have shown that our method can effectively detect and report redundancies. Our results emphasise the importance of well-formulated and accurately annotated USs in detecting redundancies. In particular, we found that the benefit parts of USs tend to be more redundant than the main parts, indicating a commonality of targets across different USs.

The redundancy analysis helps to recognise functionally identical USs and to group related USs in order to reduce the project backlog and improve project management.

Finally, we developed a framework for conflict analysis that utilises NLP techniques and VerbNet as a computational lexical resource. This method enabled us to systematically identify and report conflicts by categorising the verbs of the main parts of USs into four action annotations such as 'delete', 'create', 'forbid' and 'preserve'. Our evaluation of 19 backlog datasets revealed that high-quality USs that do not contain ambiguous verbs and nouns are crucial for effective conflict detection.

Recognising and resolving conflicts between USs ensures that project requirements are clear and coherent, preventing potential problems during implementation.

The study confirms that both syntactic and semantic analyses are essential for the management of USs in agile development. While the detection of redundancies benefits from syntactic analysis, the detection of conflicts relies heavily on semantic understanding. The quality and clarity of the annotated USs are central to the success of these methods.

Future research can build on our results by further analysing the dependencies between USs in both semantic and syntactic terms. Furthermore, we also want to compare our methods with NLP-based techniques that recognise redundancies between USs (as described in works such as [15, 22, 31]).

The insights gained from this research provide valuable guidance for improving current practices in software project management and for shaping future advances in this area.

In summary, our work contributes to streamlining software development processes by introducing robust frameworks for redundancy and conflict analysis, thereby supporting the iterative and incremental nature of agile development.

References

[1]

[2] K. AbdElazim, R. Moawad, and E. Elfakharany. A framework for requirements prioritization process in agile software development. In *Journal of Physics: Conference Series*, volume 1454, page 012001. IOP Publishing, 2020.

[3] A. Al-Hroob, A. T. Imam, and R. Al-Heisa. The use of artificial neural networks for extracting actions and actors from requirements document. *Information and Software Technology*, 101:1–15, 2018.

[4] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I 13*, pages 121–135. Springer, 2010.

[5] M. Arias, A. Buccella, and A. Cechich. A Framework for Managing Requirements of Software Product Lines. *Electronic Notes in Theoretical Computer Science*, 339:5–20, July 2018. ZSCC: 0000025.

[6] S. Arulmohan, M.-J. Meurs, and S. Mosser. Extracting Domain Models from Textual Requirements in the Era of Large Language Models. MDEIntelligence (co-located with ACM/IEEE 26th International Conference on . . . , 2023.

[7] C. F. Baker, C. J. Fillmore, and J. B. Lowe. The berkeley framenet project. In *COLING 1998 Volume 1: The 17th International Conference on Computational Linguistics*, 1998.

[8] D. M. Berry and E. Kamsties. Ambiguity in requirements specification. In *Perspectives on software requirements*, pages 7–44. Springer.

[9] K. Born, T. Arendt, F. Heß, and G. Taentzer. Analyzing conflicts and dependencies of rule-based transformations in Henshin. In *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 18*, pages 165–168. Springer, 2015.

[10] E. Cambria and B. White. Jumping NLP curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[11] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. N. och Dag. An industrial survey of requirements interdependencies in software product release planning. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 84–91. IEEE, 2001.

[12] F. Dalpiaz. Requirements Data Sets (User Stories), 2018.07.28.

[13] F. Dalpiaz, A. Ferrari, X. Franch, and C. Palomares. Natural Language Processing for Requirements Engineering: The Best Is Yet to Come. 35:115–119, 2018.

[14] J. Doe. Recommended practice for software requirements specifications (ieee). *IEEE, New York*, 2011.

[15] A. G. Duszkiewicz, J. G. Sørensen, N. Johansen, H. Edison, and T. R. Silva. On identifying similar user stories to support agile estimation based on historical data. In *1st International Workshop on Agile Methods for Information Systems Engineering, Agil-ISE 2022*, pages 21–26, 2022.

[16] A. Gomez, G. Rueda, and P. P. Alarcón. A systematic and lightweight method to identify dependencies between user stories. In *Agile Processes in Software Engineering and Extreme Programming: 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010. Proceedings 11*, pages 190–195. Springer, 2010.

[17] D. Greer and G. Ruhe. Software release planning: an evolutionary and iterative approach. 46:243–253, 2004.

[18] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[19] A. C. D. H. J. Hartmanis, T. Henzinger, J. H. N. J. T. Leighton, and M. Nivat. Monographs in Theoretical Computer Science An EATCS Series. 2006.

[20] J. H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proceedings of the 24th international conference on software engineering*, pages 105–115, 2002.

[21] P. Heck and A. Zaidman. A quality framework for agile requirements: a practitioner's perspective. *arXiv preprint arXiv:1406.4692*, 2014.

[22] M. Jurisch, S. Böhm, M. Lusky, and K. Kahlcke. Similarity measures and requirements for recommending user stories in large enterprise development processes.

[23] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. In *Model Checking Software: 13th International SPIN Workshop, Vienna, Austria, March 30-April 1, 2006. Proceedings 13*, pages 299–305. Springer, 2006.

[24] J. Kim, J. Kim, S. Park, and V. Sugumaran. A multi-view approach for requirements analysis using goal and scenario. *Industrial Management & Data Systems*, 104(9):702–711, 2004.

[25] J. Kim, M. Kim, H. Yang, and S. Park. A method and tool support for variant requirements analysis: goal and scenario based approach. In *11th Asia-Pacific Software Engineering Conference*, pages 168–175. IEEE, 2004.

[26] M. Kim, S. Park, V. Sugumaran, and H. Yang. Managing requirements conflicts in software product lines: A goal and scenario based approach. *Data & Knowledge Engineering*, 61(3):417–432, 2007.

[27] K. Kipper, A. Korhonen, N. Ryant, and M. Palmer. Extending VerbNet with Novel Verb Classes. In *LREC*, pages 1027–1032, 2006.

[28] A. Korhonen and T. Briscoe. Extended lexical-semantic classification of English verbs. In *Proceedings of the Computational Lexical Semantics Workshop at HLT-NAACL 2004*, pages 38–45, 2004.

[29] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In C. E. Brodley and A. P. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 282–289. Morgan Kaufmann, 2001.

[30] L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Huebert. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 716–727, 2018.

[31] B. Levin. *English verb classes and alternations: A preliminary investigation*. University of Chicago press, 1993.

[32] O. I. Lindland, G. Sindre, and A. Solvberg. Understanding quality in conceptual modeling. *IEEE software*, 11(2):42–49, 1994.

[33] O. Liskin, R. Pham, S. Kiesling, and K. Schneider. Why we need a granularity concept for user stories. In *Agile Processes in Software Engineering and Extreme Programming: 15th International Conference, XP 2014, Rome, Italy, May 26-30, 2014. Proceedings 15*, pages 110–125. Springer, 2014.

[34] K. Logue and K. McDaid. Handling uncertainty in agile requirement prioritization and scheduling using statistical simulation. In *Agile 2008 Conference*, pages 73–82. IEEE, 2008.

[35] G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, and S. Brinkkemper. Improving agile requirements: the quality user story framework and tool. *Requirements engineering*, 21:383–403, 2016.

[36] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper. Forging high-quality User Stories: Towards a discipline for Agile Requirements, 2015.

[37] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling*, 6:269–285, 2007.

[38] G. A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[39] M. Moens, M. Steedman, et al. Temporal ontology and temporal reference., 2005.

[40] S. Mosser, C. Pulgar, and V. Reinharz. Modelling Agile Backlogs as Composable Artifacts to support Developers and Product Owners. *Journal of Object Technology (JOT)*, 2022.

[41] E. Paja, F. Dalpiaz, and P. Giorgini. Managing security requirements conflicts in socio-technical systems. In *Conceptual Modeling: 32th International Conference, ER 2013, Hong-Kong, China, November 11-13, 2013. Proceedings 32*, pages 270–283. Springer, 2013.

[42] C. A. Prolo. Generating the XTAG English grammar using metarules. In *COLING 2002: The 19th International Conference on Computational Linguistics*, 2002.

[43] I. K. Raharjana, D. Siahaan, and C. Fatichah. User stories and natural language processing: A systematic literature review. *IEEE access*, 9:53811–53826, 2021.

[44] W. N. Robinson. Integrating multiple specifications using domain goals. *ACM SIGSOFT Software Engineering Notes*, 14(3):219–226, 1989.

[45] C. Rolland, C. Souveyet, and C. B. Achour. Guiding goal modeling using scenarios. *IEEE transactions on software engineering*, 24(12):1055–1071, 1998.

[46] S. Sachdeva, A. Arya, P. Paygude, S. Chaudhary, and S. Idate. Prioritizing user requirements for agile software development. In *2018 International Conference On Advances in Communication and Computing Technology (ICACCT)*, pages 495–498. IEEE, 2018.

[47] T. Sedano, P. Ralph, and C. Péraire. The product backlog. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 200–211. IEEE, 2019.

[48] D. Strueber. Henshin/Conflict and Dependency Analysis, 2023.02.11.

[49] M. Tichy, C. Krause, and G. Liebel. Detecting performance bad smells for henshin model transformations. *Amt@ models*, 1077, 2013.

[50] G. Varró, F. Deckwerth, M. Wieber, and A. Schürr. An algorithm for generating model-sensitive search plans for EMF models. In *Theory and Practice of Model Transformations: 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings 5*, pages 224–239. Springer, 2012.

[51] X. Wang, L. Zhao, Y. Wang, and J. Sun. The role of requirements engineering practices in agile development: an empirical study. In *Requirements Engineering: First Asia Pacific Requirements Engineering Symposium, APRES 2014, Auckland, New Zealand, April 28-29, 2014. Proceedings*, pages 195–209. Springer, 2014.

[52] Y. Wautelet, S. Heng, S. Kiv, and M. Kolp. User-story driven development of multi-agent systems: A process fragment for agile methods. 2017.

[53] Y. Wautelet, S. Heng, M. Kolp, and I. Mirbel. Unifying and extending user story models. In *Advanced Information Systems Engineering: 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings 26*, pages 211–225. Springer, 2014.

[54] M. Wynne, A. Hellesoy, and S. Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.