Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
AG Softwaretechnik

Analysis of Conflicts and Dependencies between User Stories using Graph Transformation

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science

vorgelegt von
Amir Rabieyan Nejad
Matrikel-Nr: 3350269

25. März 2024

1 Introduction

User stories (USs) are fundamentally informal units in the field of software development, which should be as precise and testable as possible descriptions of the functionality of a software in the dynamic framework of *agile development*. The iterative implementation of these user stories, especially in the context of *behaviour driven development* (BDD), plays a decisive role in the efficiency and success of a project[**?**]. As development teams navigate through the *product backlog* to prioritise and sequence the user stories, the inherent interdependencies between them become clear, presenting challenges to efficient implementation and potential conflicts.

Product backlog in agile software development acts as a repository for development tasks, reflecting the evolving needs and concerns of *system stakeholders*[**?**]. 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 artifact guiding the development process[**?**]. However, there is no formal language for expressing stories or modelling backlogs from a practical point of view. Managing dependencies between user stories is increasingly important, and although agile methodologies such as *scrum* advocate collaboration and daily stand-up meetings to resolve conflicts, the process can be resource intensive, especially with extensive backlogs.

In between automated support for extracting domain models from requirements artifacts such as USs play a central role in effectively supporting the detection of dependencies and conflicts between user stories. Domain models are a simple way to understand the relationship between artifacts 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. The objective is to shorten the feedback loop between developers and POs while supporting agile development's iterative and incremental nature[**?**].

Natural language processing (NLP) is a *computational* method for the automated analysis and representation of human language [**?**]. NLP techniques offer potential advantages to improve the quality of USs and can be used to parse, extract, or analyze user story's data. It has been widely used to help in the software engineering domain (*e.g.*, managing software requirements [**?**], extraction of actors and actions in requirement document [**?**]. Furthermore, the incorporation of computational lexicon resources like *VerbNet*[1] aids in semantic analysis, capturing linguistic and semantic data for a comprehensive understanding.

To systematically identify conflicts and dependencies between USs, the *critical pair analysis*(CPA) extension of Henshin is used to determine whether USs influence each other through model-driven transformation rules. CPA for graph rewriting [**?**] has been adapted to rule-based model transformation, *e.g.* to find conflicting functional requirements for software systems [**?**], or to analyses potential causal dependencies between model refactorings [**?**], which helps to make informed decisions about the most appro-

[1] https://verbs.colorado.edu/verbnet

priate refactoring to apply next.

This thesis attempts to introduce a well-structured workflow. The overall goal is to develop a framework that accelerates the automatic detection of conflicts and dependencies in USs expressed in natural language using graph transformation.

For this reason, we use the model presented by Mooser et al. 2022 and present an extended model, which is visualised in Figure 1 and illustrates the subsequent phases:

- Initially, Mooser et al. used the only publicly available residue dataset as the dataset**??**. They optimise the backlog and transfer it to the *Conditional Random Fields* (CRF) tool. The CRF tool is used to generate a graph-based model that represents the refined and annotated backlog to recognise *Entities*, *Actions*, *Personas* and *Benefits* of USs**??**.

- In the next step, we want to use annotated blacklog datasets generated by CRF tool and utilise the actions generated by CRF that correspond to the verbs in each user story. These actions are then processed through a *computational lexical resource VerbNet*, to categorize them into four distinct categories namely "create", "delete", "forbid" and "preserve".

- Following that, the *Henshin* tool, specifically the transformation module through Henshin's *application programming interface* (API), is employed to create a class model and formulate transformation rules aligned with each user story.

- We then use a Henshin extension known as *Critical Pair Analysis* (CPA) to automatically analyse conflicts and dependencies between user stories programmatically via their API.

- With the outcomes of the CPA analysis, the requirements engineer can visualize conflicts and dependencies between user stories in a text-based format. This presentation is designed to be lightweight and facilitates comprehension for both the development team and the client.

This thesis is structured as follows: Related work is listed in section 2. In Section 3, we present a product backlog containing a set of annotated user stories to evaluate them with the QUS framework and AQUSA. In Section 4, we implement a method to extract all verbs from the product backlog to categorise them into four categories, namely "create", "delete", "forbid" and "preserve". In Section 5, we implement a method to convert annotated user stories into transformation rules and take action on each verb category. In Section 6, we implement a method to automatically analyse conflicts and dependencies between user stories using Henshin's CPA programming interface and conclude with Section 7. [h]

2 Related Work

This section discusses relevant work such the role of USs and backlogs in agile development (Section 2.1) and the most frequently used pattern of USs is presented. In Section

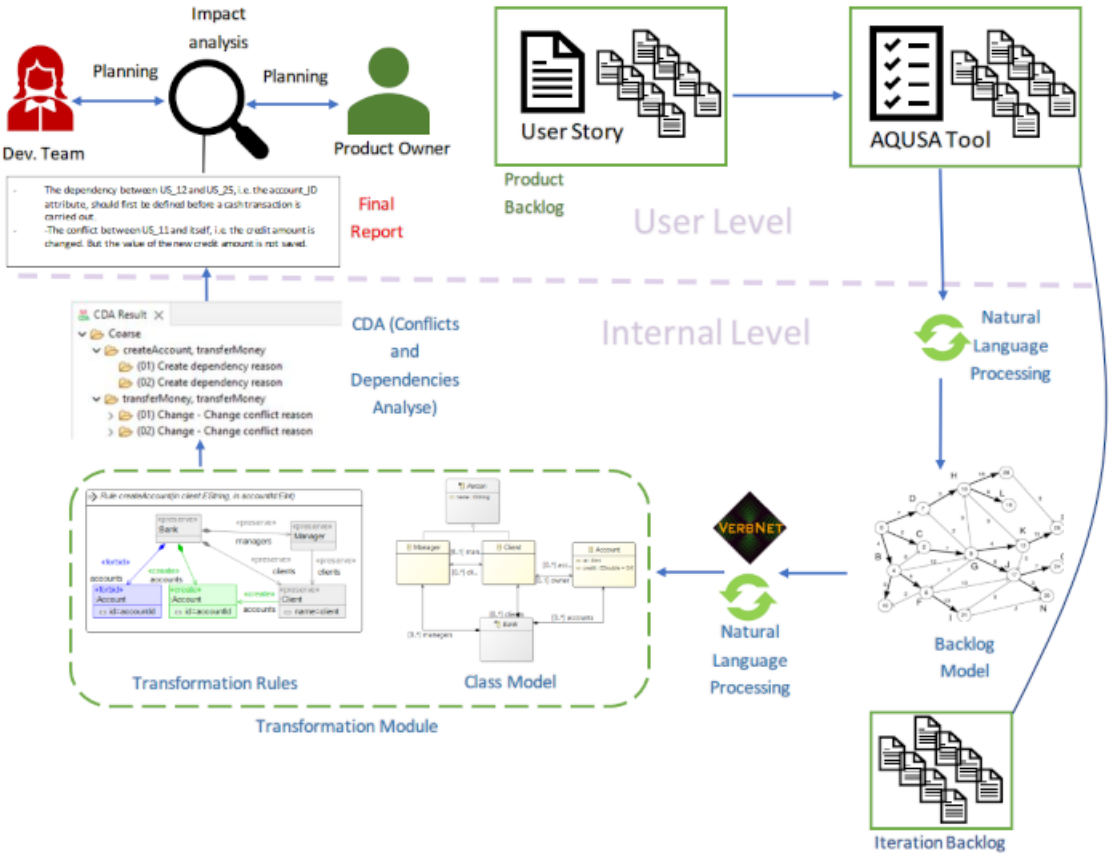

Figure 1: Extending backlog model represented by Mosser et al., 2022[**?**]

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 dependencies between user stories. 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 dependencies.

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 [**?**] became a first-class citizen during the product development.

Furthermore, thanks to a set of user stories 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" [?]. 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[2] . 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 [?]. Large, vague stories are called epics. While user stories vary widely between organizations, most observed stories included a motivation and acceptance criteria. The brief motivation statement followed the pattern: As a <*user*> 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*[?]. 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) [?].

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

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

User stories usually have dependencies, i.e. the order in which they are implemented plays a decisive role. This raises the question of how to recognise and identify the relationships between the user stories.

2.2 QUS Framework and AQUSA as a Tool

Lucassen et al. [?] 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 2 illustrates the structure of the agile requirements verification framework. Table 1 also shows the QUS framework, which defines 13 criteria for the quality of user stories.

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.

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

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 [**?**].

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

- *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:

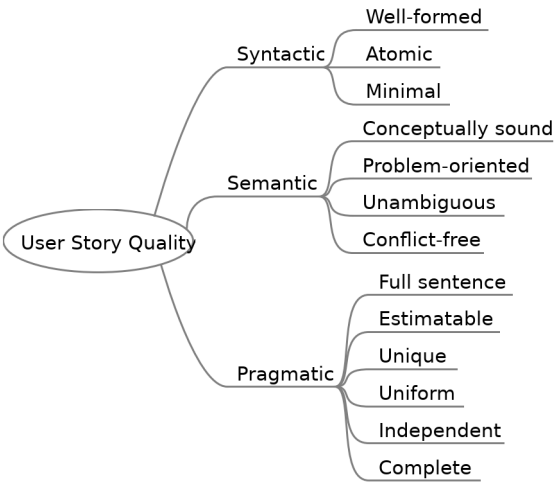

Figure 2: Agile Requirements Verification Framework [**?**]

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".

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 1: Quality User Story framework that defines 13 criteria for user story quality [**?**]

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[**?**]. 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: Mockup 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 mockup 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 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 [**?**] 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 user stories 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[3].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 user stories, 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 collection 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:

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

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

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 user stories to ensure their quality. A requirements conflict occurs when two or more requirements cause an inconsistency[**?**] [**?**]. 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". [**?**]

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 3). The set of user stories in a project is denoted by $U = (\mu_1, \mu_2, ...)$.*

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

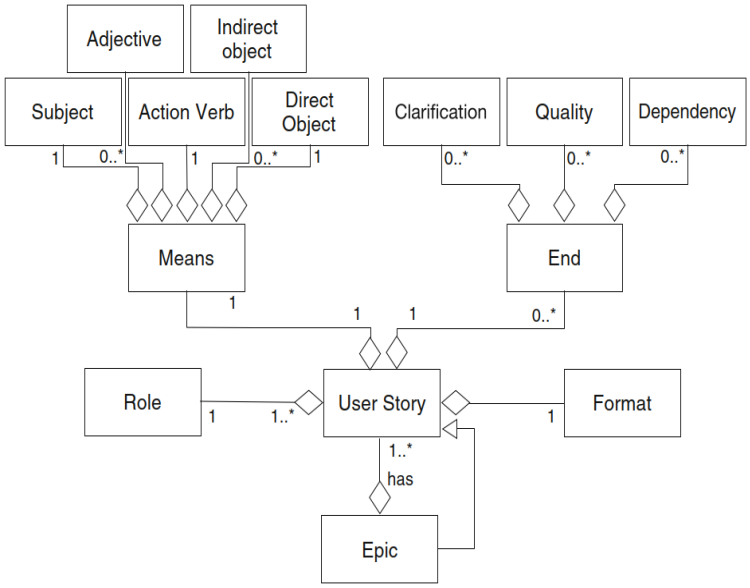

Figure 3: Conceptual model of the user stories [**?**]

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 [**?**]. 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 [**?**]. AQUSA consists of five main architectural components (Figure 4): linguistic parser, US base, analyzer, enhancer, and report generator.

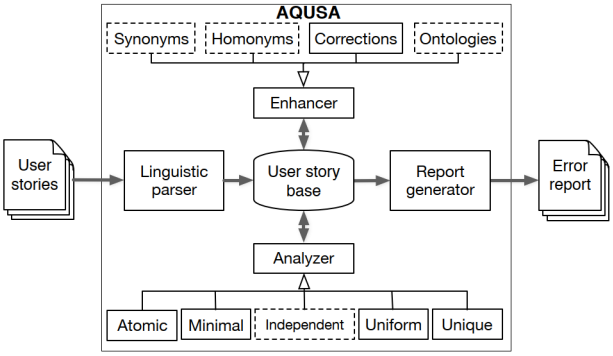

Figure 4: Functional view on architecture of AQUSA. Dashed components are not fully implemented yet [**?**]

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 [**?**].

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 verifying whether a string of text is a US is splitting it 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’ 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: Mockup 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 5.

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?

If we look at the analysis of the similarity of texts, *e.g.* with transformation tools or similar, we realize that our analysis cannot effectively recognise strong overlaps between USs and will not act precisely. We try to capture the content of the USs through the annotation just through the action, so we can do very lightweight conflict analysis similarity of texts. Very lightweight because there are many inaccuracies, as is often the case with the USs. But otherwise it is as if the analysis of two USs are similar but not the same, and somehow contradict each other.

One is the conflict that can arise in the final described system versus the inconsistencies during the specification of 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.

Similarly, with dependencies, there are explicit dependencies to define in the USs by saying this US is based on that and giving it an ID. In another case, we can analyse implicit dependencies by determining, *e.g.*, something has to be created at this position first so that it can be used in another position and so creating and using it happens in two different USs. We call this implicit dependency because it is not explicitly stated

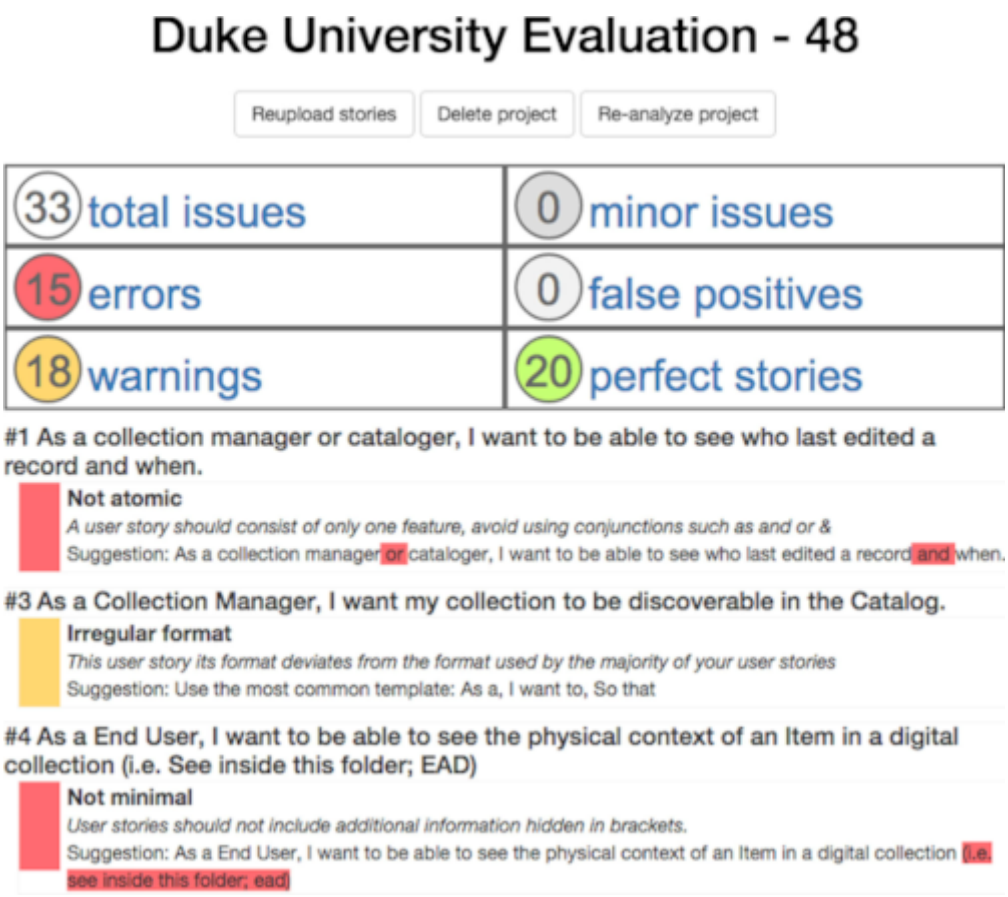

Figure 5: Example report of a defect and warning for a story in AQUSA [**?**]

in USs, but if we analyse what is there semantically, we can find out that there is a dependency.

In addition, we could use the US-ID to obtain information about the implicit dependencies between USs and compare these with the explicit dependencies to ensure that explicit and implicit dependencies are consistent with each other?

Both conflicts and dependencies 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.

Moreover, analyzing dependencies through AQUSA v1 is a component of the forthcoming effort by Lucassen et al. Which means, the automatic analysis of dependencies and conflicts between USs is an area that requires future attention and development. 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 *nlp* 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 *critical pair analysis* (CPA), which plays a central role in our methodology. 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

Automated support for extracting domain models from requirements artifacts such as USs play a central role in effectively supporting the detection of dependencies and conflicts between user stories. Domain models are a simple way to understand the relationship between artifacts and the whole system. In this subsection, we present a graph-based extraction modelling approach called CRF 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 [**?**]. 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 6. 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).

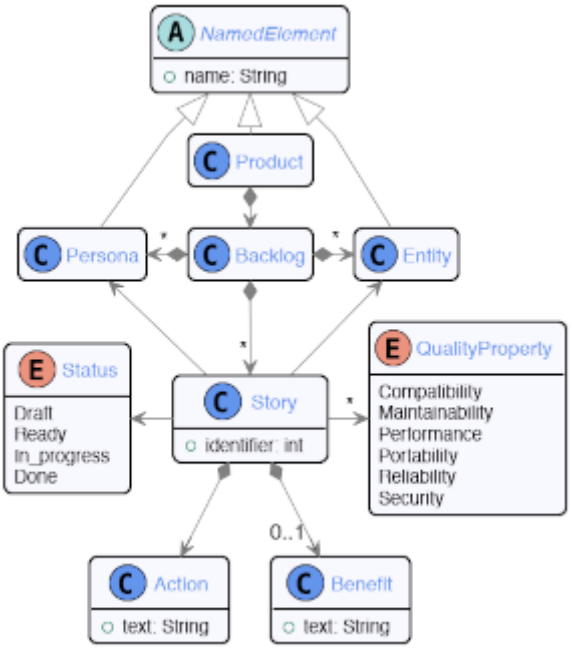

Figure 6: Backlog conceptual metamodel [**?**]

Consider, for example, the following story, extracted from the reference dataset [**?**]: "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 in the context of backlog management a system which represented in figure 7 is proposed for utilization. Building upon 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 [**?**].

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

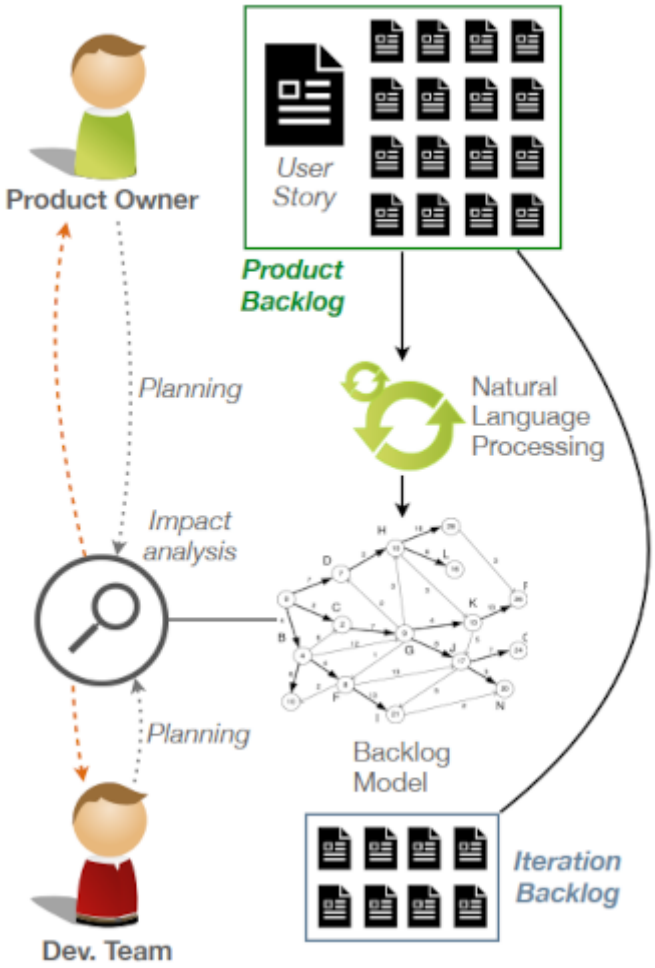

Figure 7: Providing early feedback at the backlog level [**?**]

system) Mossser et al. chose open-world(ontological) representation by modelling backlog as graphs [**?**]. 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 Storys* $\in S$ *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* $S : \forall (s, s') \in S^2, s \neq s' \Rightarrow s \not\equiv s'$.

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* [**?**].

- *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").

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

Conditional Random Fields (CRF)

CRFs [**?**] 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 [**?**].

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 [**?**]. 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*.

Mooser 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. Mooser et al. bound every *Persona* to every primary *Action* (as *trigger* relations), and every primary *Actions* to every primary *Entity* (as *target* relations) [**?**].

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 2). 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 2.

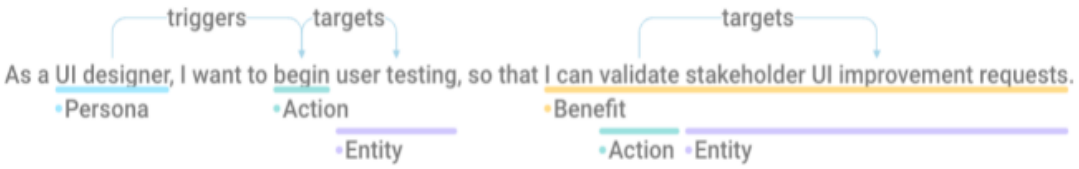

Figure 8: Example of annotated user using Doccano Annotation UI [**?**]

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

Conclusion

Conditional Random Fields (CRF) approach is graph-based and promises a high degree of precision and recall, which is particularly important in the context of domain concept extraction. CRF can cover both syntactic and semantic aspects, especially when complemented by a suitable conceptual metamodel, making them suitable for definition as a type graph in Henshin. The annotations generated by CRF can then be used for transformation into a rule-based graph transformation system, which improves support for DevOps practices.

3.2 NLP and VerbNet as a Computational Lexical Resource

Natural language processing (NLP) is a computational method for the automated analysis and representation of human language [**?**]. 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 [**?**], extraction of actors and actions in requirement document [**?**].

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 [**?**].

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, part-of-speech (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 [**?**]. 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 [**?**].

Upon the completion of the transformation of USs utilizing a Conditional Random Fields (CRF) approach, wherein entities, actions (both primary and secondary), persona, and their relational attributes (specifically, triggers, targets and contains) are meticulously annotated and structured as a graph-based representation, a preliminary imperative emerges. This imperative entails the determination of a representative semantic interpretation for the ascertained actions. This determination, in turn, serves as a prerequisite for the generation of corresponding transformation rules, namely, "create", "delete", "preserve" and "forbidden" rules.

The attainment of this representative semantic interpretation hinges upon the appli-

cation of a suite of foundational lexical resource techniques of a conceptual nature. These techniques assume a pivotal role in furnishing the essential cognitive infrastructure, facilitating a comprehensive grasp of the semantic roles, syntactic characteristics, and the systematic categorization to "creation", "deletion", "preservation" and "forbiddance" of linguistic elements embedded within the construct of the US.

VerbNet

VerbNet (VN) is a hierarchical domain-independent, broad-coverage verb lexicon with mappings to several widely-used verb resources, including WordNet [**?**], Xtag [**?**], and FrameNet [**?**]. 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 [**?**]. The original Levin classes have been refined, and new subclasses added to achieve syntactic and semantic coherence among members.

Syntactic Frames

Semantic restrictions, such as constraints related to animacy, humanity, or organizational affiliation, are employed to limit the types of thematic roles allowed within these classes. Furthermore, each syntactic frame may have constraints regarding which prepositions can be used.

Additionally, there may be additional constraints placed on thematic roles to indicate the likely syntactic nature of the constituent associated with that role. Levin classes are primarily characterized by NP (noun phrase) and PP (prepositional phrase) complements.

Some classes also involve sentential complementation, albeit limited to distinguishing between finite and non-finite clauses. This distinction is exemplified in VN, particularly in the frames for the class Tell-37.2, as shown in Examples (1) and (2), to illustrate how the differentiation between finite and non-finite complement clauses is implemented.

1. Sentential Complement (finite):
   "Susan told Helen that the room was too hot."
   Agent V Recipient Topic [+sentential – infinitival]

2. Sentential Complement (nonfinite):
   "Susan told Helen to avoid the crowd."
   Agent V Recipient Topic [+infinitival – wh_inf]

Semantic Predicates

Each VN frame also contains explicit semantic information, expressed as a conjunction of Boolean semantic predicates such as "motion", "contact", or "cause". Each of these predicates is associated with an event variable E that allows predicates to specify when in the event the predicates are true $start(E)$ for preparatory stage, $during(E)$ for the culmination stage, and $end(e)$ for the consequent stage).

Relations between verbs (or classes) such as antonymy and entailment present in WordNet and relations between verbs (and verb classes) such as the ones found in FrameNet can be predicted by semantic predicates. Aspect in VN is captured by the event variable argument present in the predicates.

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 [**?**], and class numbers 58-109, developed later by Korhonen and Briscoe [**?**]. 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 9 illustrate an example of class hierarchy from spray-9.7 class. Verb Classes are

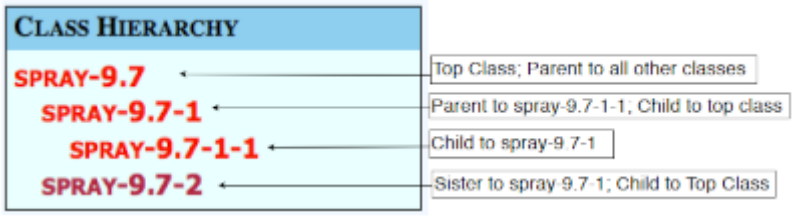

Figure 9: Class hierarchy for spray-9.7 class [**?**]

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.

Classes that share a top class can also be divided into subclasses, such as wipe verbs in wipe_manner (10.4.1) and wipe_inst (10.4.2) which specify the manner and instrument of wipe verbs in the "Verbs of Removing" group of classes (class number 10).

An example of top-class numbers and their corresponding types is given in Table 3. Class numbers 1-57 are drawn directly from Levin's (1993) classification. Class numbers 58-109 were developed later in the work of Korhonen & Briscoe (2004). Notably, the verb types of the later classes are less general, as indicated by the fact that most of these classes have a one-to-one relationship between verb type and verb class.

Conclusion

The methodological categorisation of verbs into hierarchical classes in VerbNet provides a structured and all-encompassing framework for understanding verb behaviour. This

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 3: An example of top-class numbers and their corresponding verb-type[**?**]

alignment with our project goals is of utmost importance as it supports our task of formulating transformation rules based on semantic interpretations of the actions (verbs) recognised by CRF and described in USs. VerbNet's organisational structure fits seamlessly with the requirements of our project and ensures the necessary precision and granularity that is essential for our semantic analysis and rule generation.

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 pushout approach or the single pushout approach, which are both concepts based on pushouts from category theory in the category Graphs.

In this section, some basic definitions about graph transformation and transformation rules are first established for better understanding. Afterwards, we dive into graph transformation tool Henshin [**?**], which play a pivotal 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 outlined in section 3.1, where we elucidated our utilization of conditional random fields (CRF) as a graph-based metamodeling and compositional approach for annotating USs, each US is meticulously structured and annotated in the form of a graph. Subsequently, we will apply transformation rules to these CRF-generated graphs to modelling graph transformation.

To accomplish this objective, we have harnessed the capabilities of established lexical resources such as VerbNet. This utilization of VerbNet enables us to categorize actions (which are verbs) extracted from the US into four distinct categories, namely: “create”, “delete”, “preserve”, “forbid”. These categories serve as essential components of transformation rules that articulate precise changes within the graph-based representation.

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 of these concepts, Henshin offers a visual syntax, sophisticated editing functionalities, 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 [**?**].

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 [**?**].

The performance of graph transformation-based model transformations 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 [**?**]. To be efficient, graph transformation tools usually employ heuristics such as search plans to provide good performance (e.g. [**?**]).

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 [**?**]. 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 [**?**].

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 [**?**] 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 [**?**] 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.

Analyzing Conflicts and Dependencies

The elements that comprise the system under construction interact with each other, establishing dependencies among them [**?**]. In Figure 10, *element A* requires *element B*, generating a dependency between them. Such dependencies are naturally inherited by the USs (US_i cannot be implemented until US_j is implemented).

Therefore, the natural dependencies between USs should be accepted as inevitable. In fact, only a fifth of the requirements can be considered with no dependencies [**?**]. The existence of dependencies between USs makes it necessary to have some implemented before others [**?**],[**?**],[**?**]. If the order of USs implementation does not consider these dependencies, it may have a large number of preventable refactorings, increasing the total cost of the project needlessly. Identifying beforehand the dependencies increases the ability to effectively deal with changes.

Hence, light systematic mechanisms are needed to help identify dependencies between USs [**?**]. The critical pair analysis (CPA) for graph rewriting [**?**] has been adapted to rule-based model transformation, *e.g.* to find conflicting functional requirements for software systems [**?**], or to analyses potential causal dependencies between model refactorings [**?**], which helps to make informed decisions about the most appropriate refactoring to apply next. The CPA reports two different forms of potential causal dependencies,

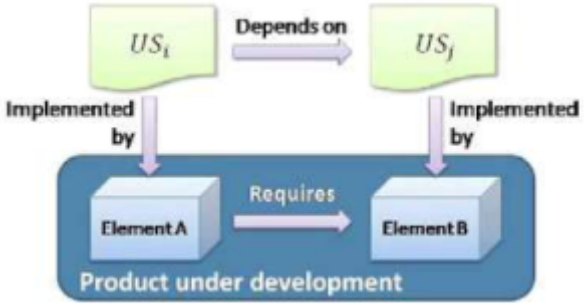

Figure 10: Inherited dependencies by user stories[**?**]

called conflicts and dependencies [**?**]. 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)**[4].

Different between Model Checking and Conflict and Dependency Analysis

In this subsection, we shall delineate two distinct analytical methodologies, specifically model checking and Conflict and Dependency Analysis. Their respective purposes are delineated in Table 4, which serves to elucidate their appropriateness for modelling USs.

Example 3.3. *Now, we exemplify conflicts and dependencies within Henshin using two US namely US_1:"As an administrator, I can add a new person to the database" and US_2: "As a visitor, I can view a person's profile"). Figure 11 delineates the class model LDAP (Lightweight Directory Access Protocol). In Figure 12, the defined rules in Henshin, specifically the rule view_profile linked to US_2, and add_person_profile corresponding to US_1, are depicted. The representation uses black to signify object preservation and green for new objects. In addition, Figure 13 (CDA result) shows the dependencies between US_2 and US_1 as* "Create dependency"*, which highlights that* Profile *node must first be created by US_1 in order to be used in US_2. Last but not least, a special instance graph is not required in Henshin.*

Example 3.4. *Looking at the USs mentioned in the example 2.1, 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."*

[4]Dependencies between rule applications can be characterized analogously.

Aspect	Model Checking for User Stories	Conflict and Dependency Analysis for User Stories
Purpose	Verify user story properties and system behavior	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 behavior	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 4: Comparative analysis between model checking and conflict and dependency methods

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

Figure 14 shows the class model "Map" while figure 15 shows the defined rules in Henshin, with each rule corresponding to a user story.

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 (Conflict and Dependency Analysis) is applied (Figure 16), 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 16 shows the conflicts and dependencies between two rules. 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 anymore.

CPA Tool

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

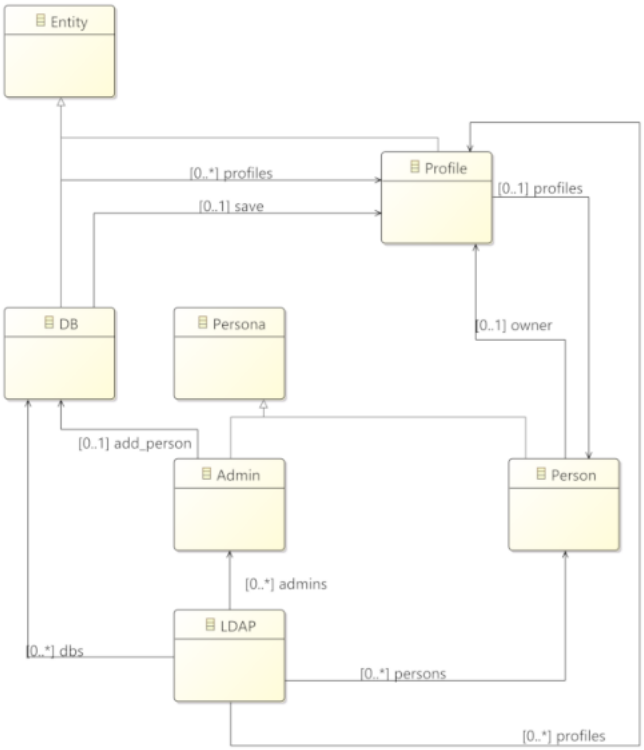

Figure 11: Henshin Class Model LDAP

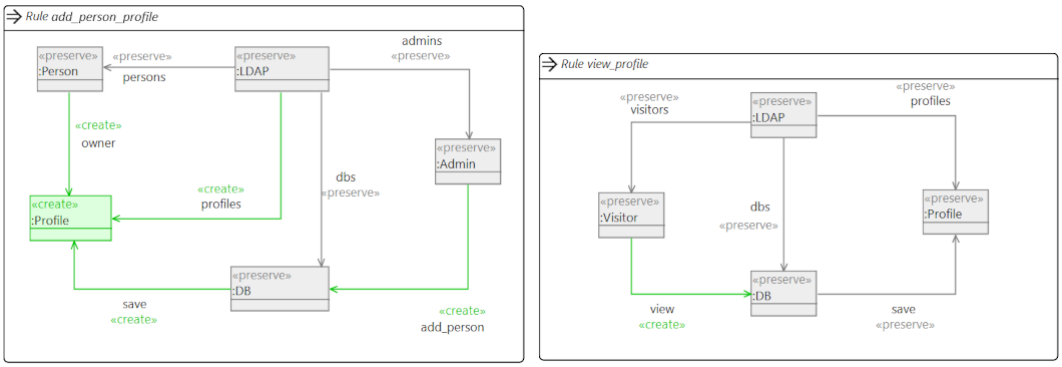

Figure 12: Illustrated rules in Henshin: view_profile rule related to US_2 and add_person related to US_1

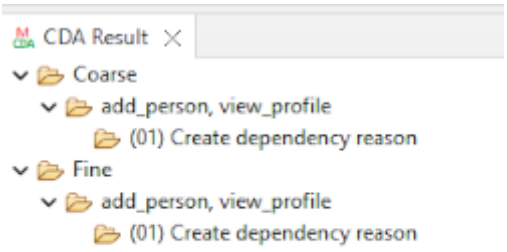

Figure 13: Henshin CDA result visualises dependencies between user stories

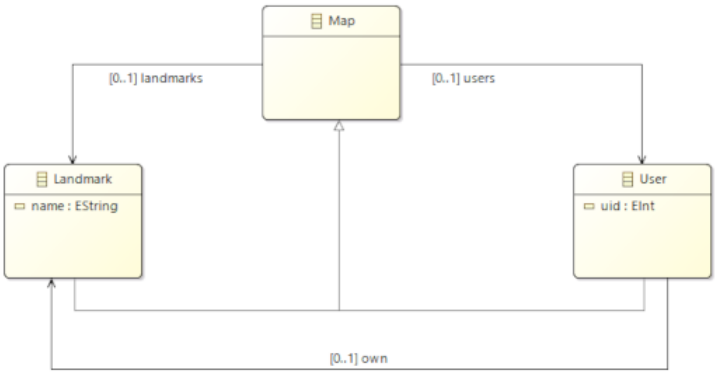

Figure 14: Henshin Class Model Map

After invoking the analysis, the rule set and the kind of critical pairs to be analyzed 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

It is notable that Henshin's critical pair analysis (CPA) remains agnostic to the specifics of instance graphs. Consequently, model checking in Henshin merely requires the set of rules. In the Henshin approach, instance graphs are not explicitly selected; instead, only rule pairs are analysed, considering conflicts and dependencies.

Since Henshin enables the specification of constraints and conditions within rules, which can be useful for enforcing and verifying US requirements to ensure that constraints are met.

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

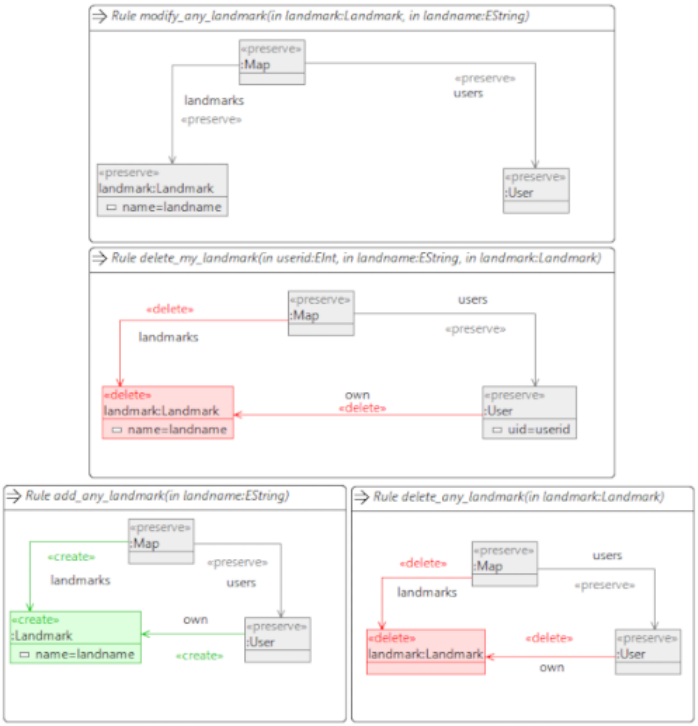

Figure 15: Illustrated rules in Henshin: add_any_landmark rule related to US_1a, modify_any_landmark rule related to US_1b, delete_any_landmark rule related to US_1c and delete_my_landmark rule related to US_2

4 Analysing Redundancy

In this Section, we present an approach for syntactically analysing redundancy using the CRF tool and Henshin.

In Section 4.1 we illustrate the requirements and functional needs that are used as input to the design phase to satisfy and bring value and benefit to the stakeholder. In the Section 4.2 we explain the design decisions of the workflow shown in 1 and we explain how the architecture is structured and what the components and their classes look like. In Section 4.3 comes the implementation to show what we have implemented and in Section **??** we show how we have tested it. Finally, we will apply this approach to 19 backlogs entered with the CRF tool[5] we enter all backlogs as input and the results are evaluated in section**??**.

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

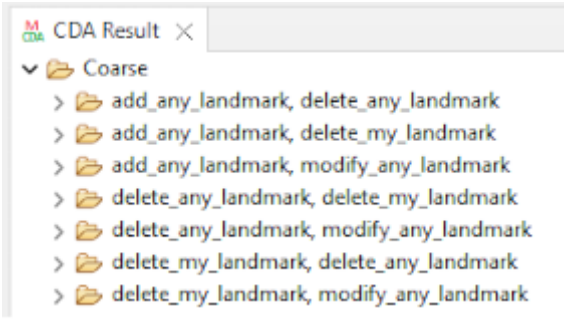

Figure 16: Henshin CPA result visualises conflicts and dependencies between user stories

4.1 Requirements

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

- As a member of a project group, I want to syntactically analyse USs that belong to a specific backlog, so that redundancies between USs can be recognise and manage accordingly.
- As a member of a project group, I would like to have a collection of US-pairs as a report that syntactically have the same clause in parts of user stories, so that I can change them if necessary.
- As a member of a project group, I want to filter the report and only see the redundancy clauses that contain "Action"(as a verb) and "Entity"(as a noun) which is called "Targets" that are duplicated in US-pair, so that I can reduce the number of potential redundancy pairs in the report.
- As a member of a project group, I want to mark founded redundancy clauses with hash symbol(#) and list those that contain "Persona"(as a noun) and "Action"(as a verb) which is called "Triggers", so that I can better see whether the Persona in is also recognised as a redundancy.
- As a member of a project group, I want to mark justified redundancy clauses with hash symbol (#) and list those that contain "Entity"(as a noun) and "Entity"(as a noun), which is called "Contains", so that I can better see whether the contained entity is also recognised as a redundancy.
- As a member of a project group, I would like to have a redundancy report that shows substantiated clauses in US-pairs and adds a hash symbol (#) at the beginning and end of the substantiated clauses as tag in each user story, so that I can see which words are duplicated in a part of the sentences.
- As a member of a project group, I want to see how many clauses are in each US-pair, so that I can aggregate each redundant US-pair based on it.

- As a member of a project group, I want a table at the top of the redundancy report that lists the US-pairs and the number of clauses contained in each pair, so that I can quickly see all the US-pairs founded and the number of clauses.
- As a member of a project group, I want to know whether the redundancy clauses belong to the main or benefit part of the US, so that I can make a decision accordingly.

4.2 Design

In this section we describing the sequence of workflow shown in 1 step-wise. Additionally, we explain what are the design decisions of this workflow and how the architecture is based on which components and classes.

To fulfil the requirements mentioned in 4.1, we use the CRF tool [**?**] for the annotation of backlogs and the Henshin API [**?**] under Java programming language for the automatic generation of rules for each US in the backlog. Moreover, we use Henshin's CDA(conflict and dependency analysis) feature [**?**] to automatically recognise conflicting US-pairs.

A detailed, step-by-step description of the workflow is given below:

CRF Tool

As input, we receive a graph-based model generated by the CRF tool, which represents the refined and annotated dataset for the recognition of *entities*, *actions*, *persons* and *benefits* of USs [**?**]. Mooser 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)[**?**]. As output we receive a JSON-file which contains all annotated USs separated by each backlogs.

Identifying USs in JSON-File

Annotated USs in each JSON-file have no identifier. To distinguish user stories, we use a Python script called *nummerize_us.py* [6], which receives JSON-files as input and adds a JSON-object called "US_Nr" with a number as value to each US and returns the JSON-files as output.

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 (Eclipse Modelling Framework). It enables the formulation of other models by utilising its constructs.

[6] https://github.com/amirrabieyannejad/USs_Annotation/tree/main/Skript/nummerize_us

Accordingly, we create an Ecore meta-model as shown in Figure 17, which is inspired by the meta-model shown in Figure 6 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.
- *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, 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*, *Edges* and annotates them with

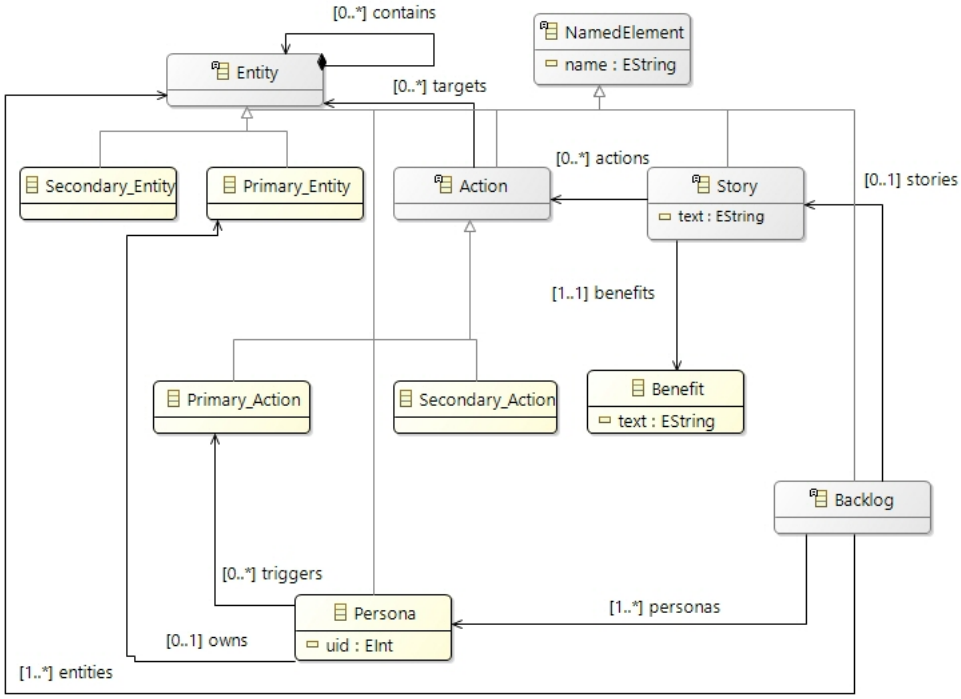

Figure 17: Ecore meta-model inspired by Mosser et al. [**?**]

<*Delete*>, <*Create*>or <*Preserve*>, which are crucial for the CDA tool to recognise the redundant pairs. To generating rules we create a package named *package org.henshin.backlog.code.rule* and specially the class *RuleCreator* which used following classes[7]:

- *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. To create a node within a transformation rule, the CRule class is required. The default action when specifying a node is the <*preserve*>action. We can also specify a different action when we create a node, for example <*delete*>or <*create*>.
- 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

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

as input and a Henshin-file containing all rules as output, where each rule and its elements 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 attributes and edges are annotated to apply CDA to Henshin-files. We decided to annotate the "name" attribute of all Primary/Secondary Actions/Entities and their associated edges including targets, triggers and contains in <delete >action. The main goal is to increase the probability of finding the US-pairs that have the same Action/Entity and target the same Entity, which can be a potentially redundant US-pair.

Methods of the RuleCreator Class

In this subsection, the method of the RuleCreator class is described as follows:

1. readJsonArrayFromFile: This method receives a JSON file as input and reads the JSON file, tokenises the JSON content and parses the JSON content into a JSON array and returns the parsed JSON-array.

2. assingCmodule: 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.

3. processJsonFile: It takes parsed JSON array as input and processes their attributes, such as persona, actions/entities, entities, text and their edges, such as targets, triggers. Corresponding elements are created as output in a the Henshin transformation module (CModule).

4. 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.

5. 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.

6. 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.

7. 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.

8. checkEntityIsTarget: It receives the name of the entity and the JSON-array with information about target 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, it returns true to indicate that the entity is a target.

9. 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 target 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 relationship is also a target of another entity.

10. processContainsEdges: It receives the JSON-object to be processed, the JSON array with information about contains/target 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 name “contains”. If one of the entities is a target of another entity (as specified in the targets array), the edge is annotated for deletion. If none of the entities is a target, the edge is annotated as “preserve”.

Conflicting and Dependency Analysis

After the henshin file has been automatically generated through the RuleCreator class, we can now pass it to the conflict and dependency analysis (CDA) to find potential redundancy pairs.

Since the CPA API [8] does not yet take into account conflicts and dependencies related to *attribute*, we decided to use the user interface (UI) of the CPA extension of Henshin , which supports domain experts in the development of rules through the interactive use of CPA.

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 and the type of analysis. We then select as *First* and *Second Rules* all US rules and as the type of analysis we select *conflicts* as illustrated in Figure 19 On the next page

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

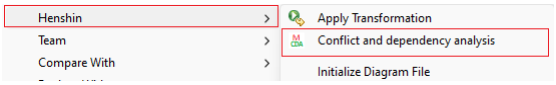

Figure 18: Applying CDA to the selected Henshin file

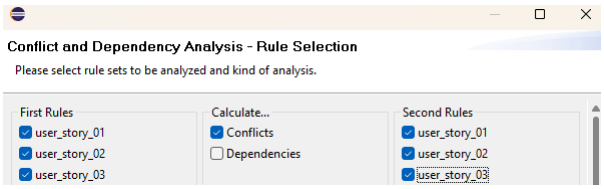

Figure 19: CDA user interface: Selection of rules and type of analysis

of the CDA user interface 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*. We choose Fine granularity as the depth of analysis due to the fact that it shows all conflict reasons for each conflicting rule pair. A conflict reason is a model fragment whose presence leads to a conflict. General conflict reasons result from different combinations of minimal conflict reasons[**?**].

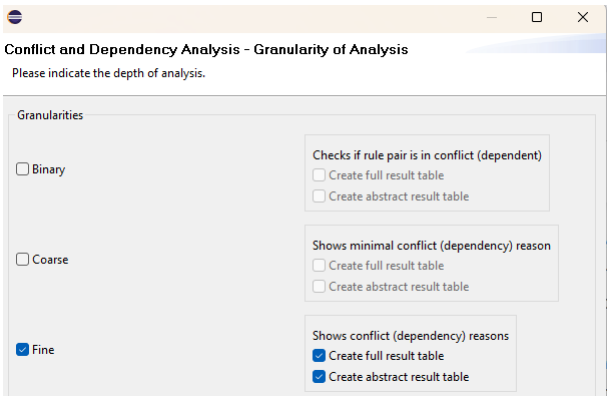

Figure 20: CDA user interface: Selection of report granularity

During the execution of the CDA analysis, the rule pair 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 conflicts.

Figure 22 shows how the data is saved in the project tree view. The results folder is created in the folder 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 reasons and atoms together in a rule pair folder.

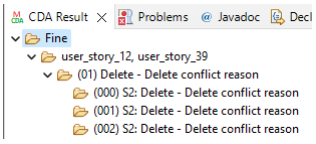

Figure 21: CDA report with fine granularity

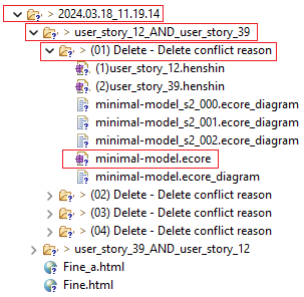

Figure 22: Saving CDA results data in the project structure view

For each conflict reason, there is a *minimal-model.ecore* 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 conflicting 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.

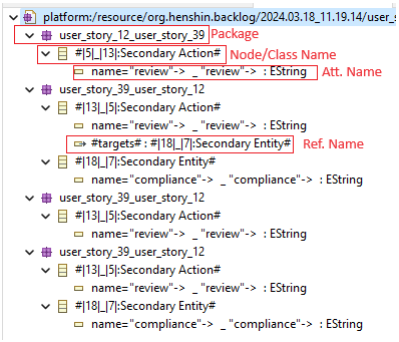

Figure 23: Representation of conflicting attributes and references in *minimal-model.ecore* file

Extracting Textual Report

To make the CDA report lightweight for the concerned group or individual, we need to extract the key information such as redundancy US-pair, redundancy clauses, number of redundancy clauses in each part or sentence (main or benefit part) and create a report

as a text file containing 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
- Text of US-pairs whose redundancy words are marked with a hash symbol (#).
- Parts of the sentence in which words and clauses are found.

Listing 1 illustrates the format 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
us_12  0      1
us_39  1      0

-----------[Potential Redundant User Stories found]-----------
{user_story_12_AND_user_story_39}

Redundant clauses within user stories are:
* Secondary Action: review
* Secondary Entity: compliance
* 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:

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 1: Example of generated textual report for two US-pair

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 form the package *org.eclipse.emf.ecore*[9]. 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 you of 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.
- 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

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 E-Attribute model object with the methods *getType* to retrieve the saved EClass and *getName* to retrieve the saved E-Attribute.

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

- PrimaryEntity/SecondaryEntity: Which has only saved the EClass specified by "Primary/Secondary Entity" and the E-Attribute model object with the methods *getType* to retrieve the saved EClass and *getName* to retrieve the saved E-Attribute.
- Targets: The EClass specified by "Primary/Secondary Action" as *outgoing edge* and an E-Attribute model object as *incoming edge* with "Primary/Secondary Entity". The method *getType* retrieve the stored EClass and method *getName* retrieve the stored E-Attribute.
- Contains: The EClass specified by "Primary/Secondary Entity" as *outgoing edge* and an E-Attribute model object as *incoming edge* with "Primary/Secondary Entity". The method *getType* retrieve the stored EClass and method *getName* retrieve the stored E-Attribute.
- Triggers:
- Contains: The EClass specified by "Persona" as *outgoing edge* and an E-Attribute model object as *incoming edge* with "Primary Action". The method *getType* retrieve the stored EClass and method *getName* retrieve the stored E-Attribute.
- RedundantPair: Stores the identifier of the two created user stories as a redundant pair. It also saves the total number of redundancy clauses within the US-pair.
- 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 calss RedundancyItems contains following methods which are crucial 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 user stories 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 user stories 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 target pair 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 generating a report of redundancy items based on the data stored in the class instance. It takes several parameters, including a FileWriter for writing to a file, lists of various pairs of targets, contains, and triggers, and a JSONObject for storing the report data in JSON format which is crucial for evaluation.

ReportExtractor Class: Methods related to Extracting Report

In this subsection, the method of the ReportExtractor class is described as follows:

1. extractReports: This method orchestrates the extraction and analysis of redundancy reports from a directory containing US conflict pairs and their associated reasons created by CDA tool, and generates both text and JSON reports for further investigation and processing.

   It receives two FileWriter objects as input, one for writing text reports and one for JSON reports.

   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.

   For each valid conflict pair directory, iterates through the conflict reason directories within the current conflict pair directory and uses the *minimalEcoreExist* method to check whether a minimal ECore file exists for a conflict reason.

   To identify redundancy items, it uses the method *processMinimalModels* and reads a 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 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 and targets. 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 text report file. It also extracts and saves relevant information in JSON format for further analysis.

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

2. 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.

3. 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.

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

   It receives a conflict pair and a conflict reason as input. 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 file exists, the method returns true, indicating that the minimal-model.ecore file exists for the specified conflict pair and conflict reason. If the file does not exist, it also returns false.

5. containsAnd: This method ensures that the folder name is identified with *_AND_*, as the report generated by CDA is formatted like "user_story<digit >_AND_ user_story<digit >". It return true if the folder contains "and", otherwise it return false.

6. iteratePackages:This method identifies redundancies within the attributes and references of EClasses in a minimal model, saves 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 EClass "#" is present; if this is the case, the conflict was detected by CDA. If an attribute is found, the class of the conflicting attribute is determined and added to the corresponding section within RedundancyItems (e.g. Primary/Secondary Action/Entity).

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

7. 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 this 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.

8. hasActions: This method is useful for using the contents of a RedundancyItem to determine whether certain actions are present in a list of founded redundant entries, in order to check later whether founded clauses contain at least one other action.

   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.

9. 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.

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

    It receives as input an array list with the names of the redundant elements and an object of the type RedundancyItems, which contains a collection of targets.

    The method checks whether there is a match with the name of any target stored in the RedundancyItems object. If a match is found between the elements, the method immediately returns true, which means that at least one target is present.

11. readJsonArrayFromFile: This method provides the ability to read JSON data from a file and convert it into a JSONArray 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.

12. 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.

13. 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.

    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.

14. 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 yes, true is returned.

15. applyHashSymbolPersona: This method identifies common triggers pairs between two segments of USs, marks them with hash symbols and returns the changed text segments together with the number of redundant trigger pairs.

    As input, it receives a list of common trigger pairs between the USs, RedundancyItems and the segments of the USs. It iterates through the list of common trigger pairs and checks whether both elements of the trigger pair are present in both segments.

    It then increments the redundancy count to keep track of the number of redundant trigger pairs. The output returned is the text of USs containing the modified text segments with hash symbols and the redundancy count.

16. applyHashSymbolTargets: This method identifies common target pairs between two segments of USs, marks them with hash symbols and returns the changed text segments together with the number of redundant target pairs.

    As input, it receives a list of common target pairs between the USs, RedundancyItems and the segments of the USs.

    It iterates through the list of common target pairs and checks whether both elements of the target pair are present in both segments. It then increments the redundancy count to keep track of the number of redundant target pairs. The output returned is the text of USs containing the modified text segments with hash symbols and the redundancy count.

17. applyHashSymbolContaians: This method identifies common contain pairs between two segments of USs, marks them with hash symbols and returns the changed text segments together with the number of redundant contain pairs.

    As input, it receives a list of common contain pairs between the USs, RedundancyItems and the segments of the USs. It iterates through the list of common contain pairs and checks whether both elements of the contain pair are present in both segments.

    It then increments the redundancy count to keep track of the number of redundant contain pairs. The output returned is the text of USs containing the modified text segments with hash symbols and the redundancy count.

18. highlightRedundancies: This method identifies redundancies between US-pair, applies hash symbols 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 sentence that occurs in the same segment (e.g. only main or only benefit part of the sentence).

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

19. writeUsText: This method reads the text of two user stories, highlights redundants between them, writes the highlighted text to a file, and records information about the redundants.

    As input it receive US-pair, array list of containing redundanted elements, a list of redundant pairs to store redundant pairs, an object type RedundancyItems containing redundancy items and a FileWriter object used for writing output.

    It extract the names 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, it sets the redundant pair and count of redundancy clauses.

20. splitUsText: 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.

21. writeUsSentencePart: This method facilitates the extraction and storage of highlighted sentence parts from USs in a file for further analysis or as a reference.

    As input, it receives the US-pair, RedundancyItems and a FileWriter allowing the extracted sentence parts to be written. It receives the text of the USs from the reundancyItems object and calls the *splitUsText* method to split the US texts into sentence parts with highlighted elements. The extracted sentence parts are also written to the file using the FileWriter. Finally, the extracted sentence parts are saved in a JSON object for further processing and analysis.

22. 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.

ReportExtractor Class: Methods related to Creating Table

The following methods in the ReprortExtractor class are responsible for creating a suitable table at the beginning of the text report:

1. writeTable: This method is used to write a table of potential redundancies between USs and the count of their total redundant elements, including the addition of the count of main and benefit redundancy elements using *createTable* method.

    As input, it receives a file object into which the table is inserted and a list of redundancy pairs objects containing information about pairs of redundant elements in USs.

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

2. 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 pairs of redundant elements in USs, where the first and second USs are to be compared.

It iterates through each RedundantPair object in the RedundantPairs list in a loop and checks for each RedundantPair object whether the pair of USs matches. If a matching pair is found, the maximum redundancy number stored in this RedundantPair object is returned. If no matching pair is found, 0 is returned, indicating that there are no redundancies between the specified US-pairs.

3. createTable: This method prepares the content for the table in which potential redundancies between user stories are displayed, taking into account the total redundancy count between each pair of user stories 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 pairs of redundant elements in USs.

   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 pair of USs 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.

4.3 Implementation

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

The entire implementation is available in the GitHub repository [10].

Objective and Scope

The goal and scope of the work is divided into three phases. Firstly, converting the USs annotated by the CRF tool into graph transformation rules; secondly, using the CDA function of the Henshin to automatically report redundancy between USs; thirdly, extracting important information from the CDA report into a text report. For further analysis, we stored the information in a JSON file to be able to import the data into another platform such as MS Excel. Figure 24 illustrates the implementation phases mentioned.

[10] https://github.com/amirrabieyannejad/USs_Annotation/tree/main

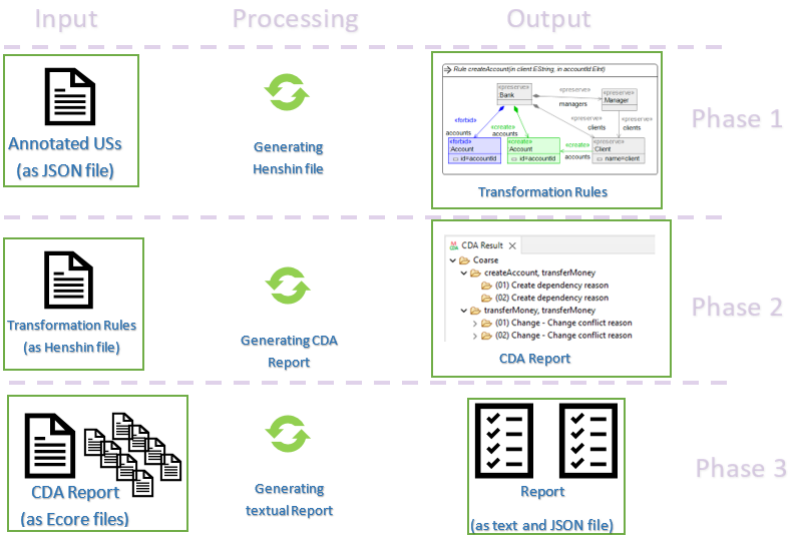

Figure 24: Three implementation phases

Methodology

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

- Eclipse as IDE: 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.
- Eclipse Modeling Project[11]: The Eclipse Modeling Project 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 Modeling Framework (EMF)[12]: 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[13]: Henshin is an in-place model transformation language for the Eclipse Modeling Framework (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)

[11] https://eclipse.dev/modeling/
[12] https://eclipse.dev/modeling/emf/
[13] https://wiki.eclipse.org/Henshin

- Henshin's CDA feature[14]: Henshin's conflict and dependency analysis feature enables the detection of potential conflicts and dependencies of a set of rules.
- GitHub as version control: 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.

Data Structures

Error Handling

Limitations

Following limitation which at the beginning should be clarify:

- We have to use Eclipse version 2023-03 because Henshin files cannot be installed with the latest version of Eclipse.
- We have to work with Java because all Henshin files APIs are not available in other programming languages like Python.
- CDA API is not yet implemented for considering relationships and dependencies between attributes. This forces us to use the graphical CDA interface 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.

4.4 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[15] as version 4 is more suitable and compatible with Eclipse version 2023-03.

[14] https://wiki.eclipse.org/Henshin/Conflict_and_Dependency_Analysis
[15] https://junit.org/junit4/

Test Environment Configuration

In the main project *org.henshin.backlog*, we create a separate package called *org.henshin.backlog.test*, which contains two Java classes *ReportExtractorTest.java* and *RuleCreator_Test.java*, each of which corresponds to the corresponding Java source code.

We defiIn the main project *org.henshin.backlog*, we create a separate package called *org.henshin.backlog.test*, which contains two Java classes *ReportExtractorTest.java* and *RuleCreator_Test.java*, each of which corresponds to the corresponding Java source code.

Scope of Testing

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

Test Cases

This section describes 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. Table 5 illustrates the test cases for the class RuleCreator_Test.java and table 6 illustrates the test case for the class ReportExtractor.java.

Test Case	Supplied Data	Expected Outcome	Description
testAssignCmodule	assing a dummy ECore model	Through an exception: *EcoreFileNotFound.class*	Check whether the ECore model already exists and CModule is correctly assigned
testProcessContainsEdges (UndefindedEntity)	Specify an entity that is not contained in the JSON file "Entity", but appears in the JSON file as "Contains"	Through an exception: *EntityInJsonFileNotFound.class*	Check whether the entity appearing in *contains* has already been identified/proceeded as an entity
testprocessJsonFile (ActionNotExist)	Specify an action that is not contained in the JSON file *Action*, but appears in the JSON file as *Contains, Targets or Triggers.*		Through an exception: *ActionInJsonFileNotFound.class*

Table 5: Test cases for RuleCreator class

Test Case	Supplied Data	Expected Outcome	Description
testEmptyDirectroy	Am empty directory	Through an exception: *CdaReportDirIsEmpty.class*	Check if CDA Report directory is empty
testEmptyJSONFile	Addressing an empty JSON file as dataset	Through an exception: *EmptyOrNotExistJsonFile.class*	Check if JSON dataset file is already existed and is not empty
Purpose	Verify user story properties and system behavior	Understand dependencies and interactions between user stories	d
Method	Large state spaces exploration	Rule-based model transformation	d
Automated vs. Manual	Automated	Automated	d
Scope	Ensuring user stories meet specified requirements and system behavior	Understanding how user stories relate to each other, managing dependencies	d
Use Cases	Ensuring user story correctness and system behavior	Agile development, impact analysis, and managing user story dependencies	d
Result	Verification of user story properties (e.g., acceptance criteria)	Identification of user story dependencies, potential conflicts, and their impact on the development process	d

Table 6: Test cases for ReportExtractor class