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

4. Februar 2024

Selbstständigkeitserklärung

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

Marburg, den 4. Februar 2024

Abstract

User stories, the basic building blocks of software development, serve as precise and testable descriptions of a software's functionality. Within the dynamic framework of *agile development*, these user stories are usually written informally in simple text and managed in the product backlog, which serves as a repository for prioritising and tracking development tasks.

Behaviour Driven Development (BDD), a special approach in the field of agile software development, places a strong emphasis on the iterative implementation of user stories. The sequence of user stories in BDD is a central aspect of the methodology. The correct sequence not only has an impact on the efficiency of development, but also on the overall success of the project. By effectively prioritising and sequencing user stories, development teams can deliver incremental value to users, respond to changing requirements and ensure that the most critical features are addressed first.

User stories often have dependencies on each other, leading to potential conflicts if one user story requires the deletion of a component that is essential for the successful execution of another user story, or if one user story introduces an element that runs counter to the realisation of another user story and thus prevents it. In addition, changing an existing requirement or adding a new requirement to the existing *product backlog* in agile software development can also cause conflicts due to changes in the needs and concerns of *system stakeholders*. This can result in a wide range of inconsistencies due to conflicting requirements, as requirements are raised by multiple stakeholders involved in product development to achieve different functions.

Normally, to minimise the occurrence of conflicts, teams should systematically identify and document dependencies between user stories within their backlog. Agile methods such as *Scrum* encourage cross-functional collaboration and daily stand-up meetings as mechanisms to address and mitigate dependencies and conflicts in a timely manner. However, this approach can be time and resource consuming. In cases where the backlog is very extensive, recognising existing conflicts between user stories can become a complex undertaking.

Model-based software engineering is a suitable method for coping with the ever-increasing complexity of software development processes. *Graphs* and *graph transformations* have proven to be useful for visualising such models and their changes.

Since there is no process in agile software development to systematically identify and manage conflicts between requirements created in natural language, this thesis aim to presents a well-structured workflow that uses a collection of techniques and tools from different domains to accelerate the automatic detection of conflicts and dependencies in natural language user stories.

Zusammenfassung

User Stories, die Grundbausteine der Softwareentwicklung, dienen als präzise und testbare Beschreibungen der Funktionalität einer Software. Innerhalb des dynamischen

Rahmens der *agilen Entwicklung* werden diese User Stories in der Regel informell in einfachem Text verfasst und im *Product Backlog* verwaltet, das als Repository für die Priorisierung und Verfolgung von Entwicklungsaufgaben dient.

Behaviour Driven Development (BDD), ein spezieller Ansatz im Bereich der agilen Softwareentwicklung, legt einen starken Schwerpunkt auf die iterative Umsetzung von User Stories. Die Reihenfolge der User Stories im BDD ist ein zentraler Aspekt der Methodik. Die richtige Reihenfolge hat nicht nur Auswirkungen auf die Effizienz der Entwicklung, sondern auch auf den Gesamterfolg des Projekts. Durch eine effektive Priorisierung und Abfolge der User Stories können die Entwicklungsteams den Benutzern einen inkrementellen Mehrwert bieten, auf sich ändernde Anforderungen reagieren und sicherstellen, dass die kritischsten Funktionen zuerst behandelt werden.

User Stories sind oft voneinander abhängig, was zu potenziellen Konflikten führen könnte, wenn eine User Story die Löschung einer Komponente erfordert, die für die erfolgreiche Ausführung einer anderen User Story unerlässlich ist, oder wenn eine User Story ein Element einführt, das der Realisierung einer anderen User Story zuwiderläuft und diese somit verhindert. Darüber hinaus kann auch die Änderung einer bestehenden Anforderung oder das Hinzufügen einer neuen Anforderung zum bestehenden Product Backlog in der agilen Softwareentwicklung zu Konflikten führen, da sich die Bedürfnisse und Anliegen der Systemstakeholder ändern. Dies kann zu einer Vielzahl von Inkonsistenzen aufgrund widersprüchlicher Anforderungen führen, da Anforderungen von mehreren an der Produktentwicklung beteiligten Stakeholdern gestellt werden, um unterschiedliche Funktionen zu erreichen.

Um das Auftreten von Konflikten zu minimieren, sollten Teams in der Regel systematisch Abhängigkeiten zwischen User Stories innerhalb ihres Backlogs identifizieren und dokumentieren. Agile Methoden wie *Scrum* fördern die funktionsübergreifende Zusammenarbeit und tägliche Stand-up-Meetings als Mechanismen, um Abhängigkeiten und Konflikte zeitnah anzugehen und zu entschärfen. Dieser Ansatz kann jedoch zeit- und ressourcenaufwendig sein. In Fällen, in denen das Backlog sehr umfangreich ist, kann das Erkennen bestehender Konflikte zwischen User Stories zu einem komplexen Unterfangen werden.

Modellbasiertes Software-Engineering ist eine geeignete Methode, um die ständig steigende Komplexität von Softwareentwicklungsprozessen zu bewältigen. *Graphen* und *Graphtransformationen* haben sich als nützlich erwiesen, um solche Modelle und deren Änderungen zu visualisieren.

Da es in der agilen Softwareentwicklung keinen Prozess zur systematischen Identifizierung und Verwaltung von Konflikten zwischen in natürlicher Sprache erstellten Anforderungen gibt, soll in dieser Arbeit ein gut strukturierter Arbeitsablauf vorgestellt werden, der eine Sammlung von Techniken und Werkzeugen aus verschiedenen Bereichen verwendet, um die automatische Erkennung von Konflikten und Abhängigkeiten in natürlichsprachlichen User Stories zu beschleunigen.

Contents

1 Introduction **8**

2 Related Work **9**
2.1 Role of User Stories and Backlogs in Agile Development 10
2.2 QUS Framework and AQUSA as a Tool 11

3 Preliminaries **23**
3.1 Extracting Domain Models from Textual Requirements 23
3.2 NLP and VerbNet as a Computational Lexical Resource 28
3.3 Analysis by Graph Transformation Tool 33

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 in Section 2.1. In addition, the most frequently used pattern of USs is presented.

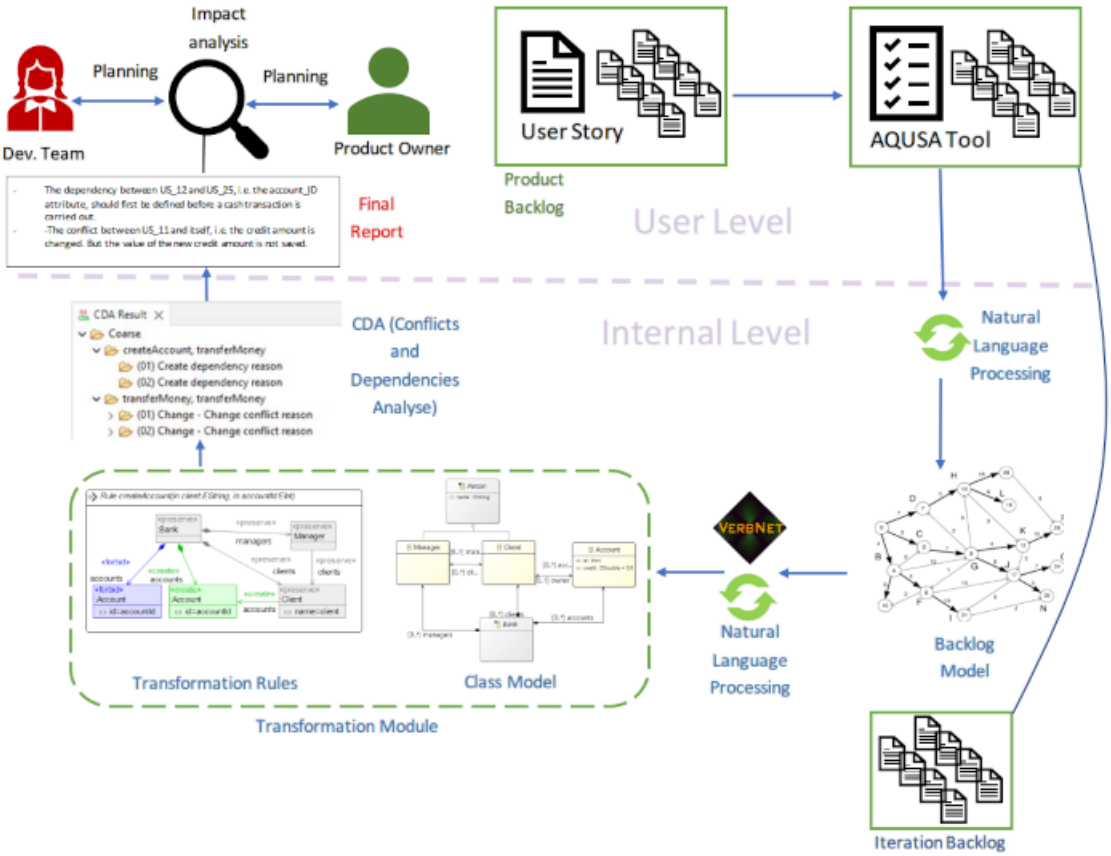

Figure 1: Extending backlog model represented by Mosser et al., 2022[**?**]

In Section 2.2 the focus is on the *quality user story* (QUS) framework and their their tool "AQUSA" to answer the question of the existence of criteria and automatic way for managing and identifying conflicts and dependencies between user stories. Following this investigation, we draw a conclusion as to whether the existing criteria are able to optimise 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 sometimes called *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, which means that the order in which they are implemented plays a decisive role. This raises the question of how to recognise and identify the relationships between 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 and divides them into two categories, namely *Individual*, which applies to single US, and *Set*, which applies to a bundle of USs. Based on QUS, Lucassen et al. present the automatic quality user

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

story artisan (AQUSA) software tool for assessing and enhancing US quality automatically. Relying on NLP techniques, AQUSA detects quality defects and suggests possible remedies.

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

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.

First, Lucassen et al. introduced quality criteria that can be evaluated against an individual US by presenting an explanation of the criterion as well as an example US that violates the specific criterion.

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

Independent

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

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 [**?**]

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

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

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".

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

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

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 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 $U = (\mu_1, \mu_2, ...)$.

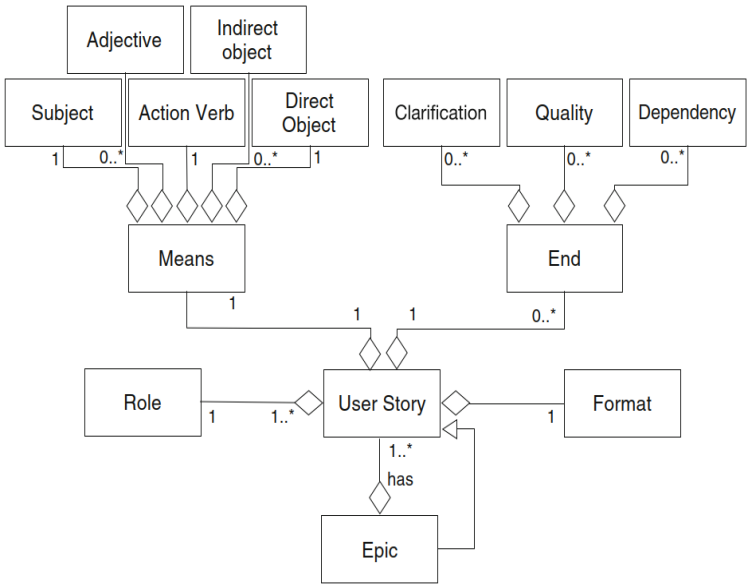

Figure 3: Conceptual model of the user stories [**?**]

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".

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.

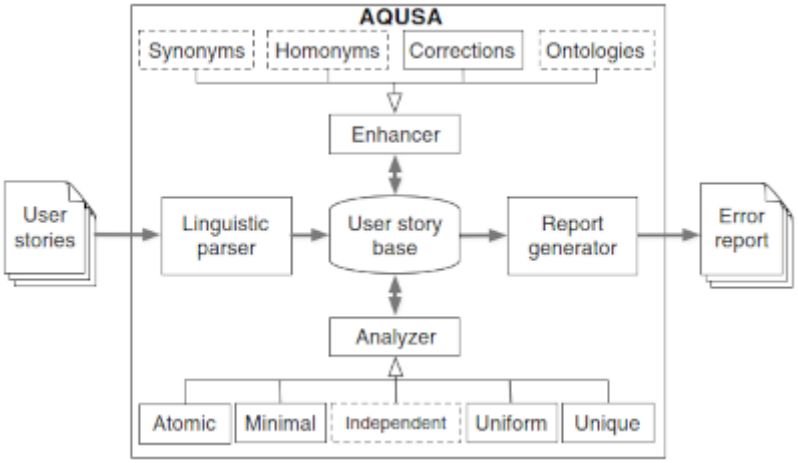

Figure 4: Functional view on architecture of AQUSA. Dashed components are not fully implemented yet [**?**]

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.

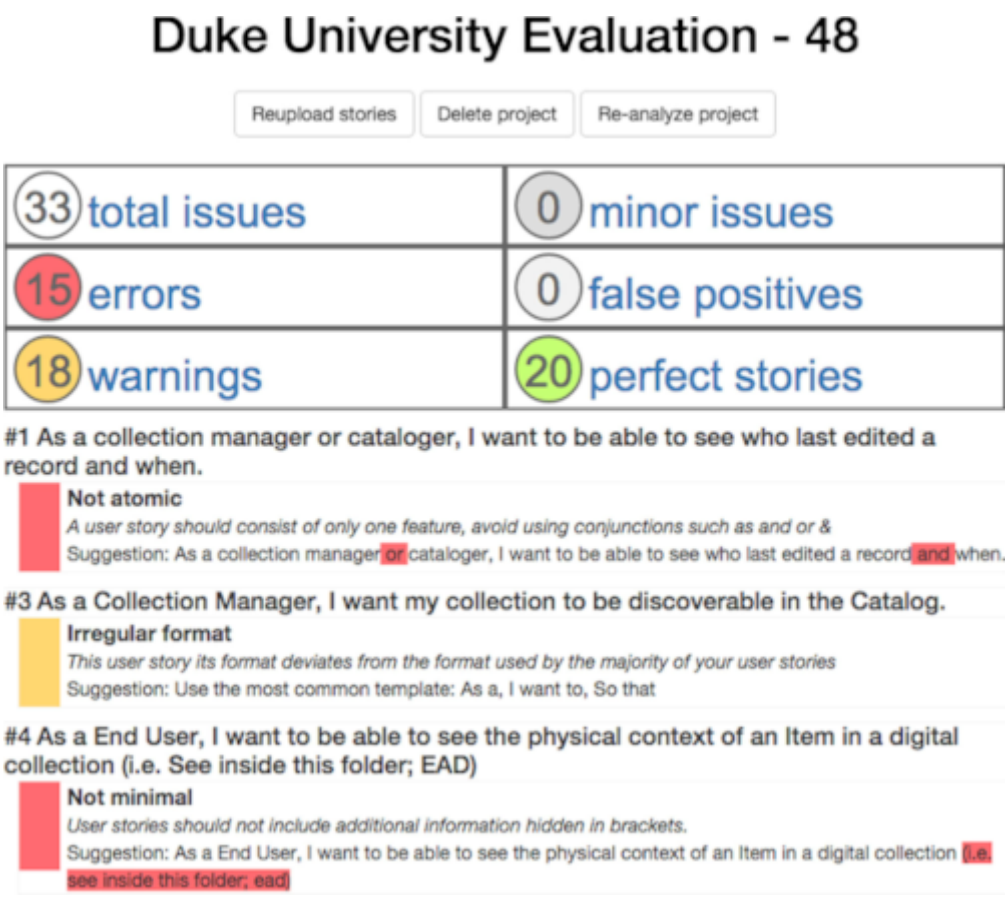

Figure 5: Example report of a defect and warning for a story in AQUSA [**?**]

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 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).

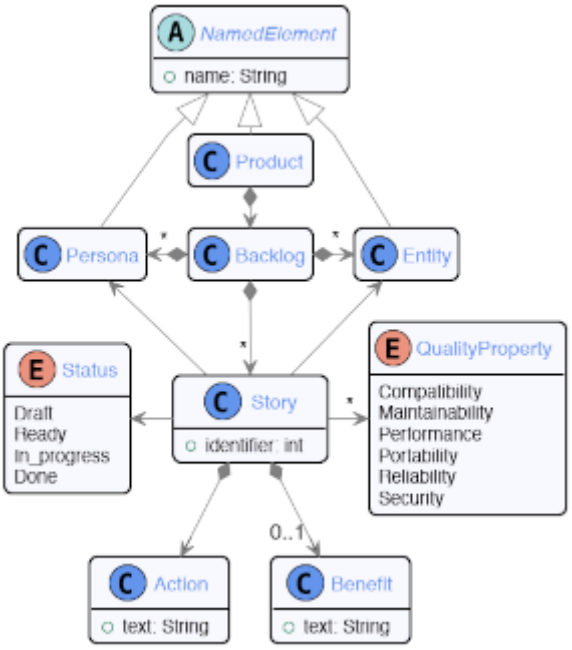

Figure 6: Backlog conceptual metamodel [**?**]

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 [**?**].

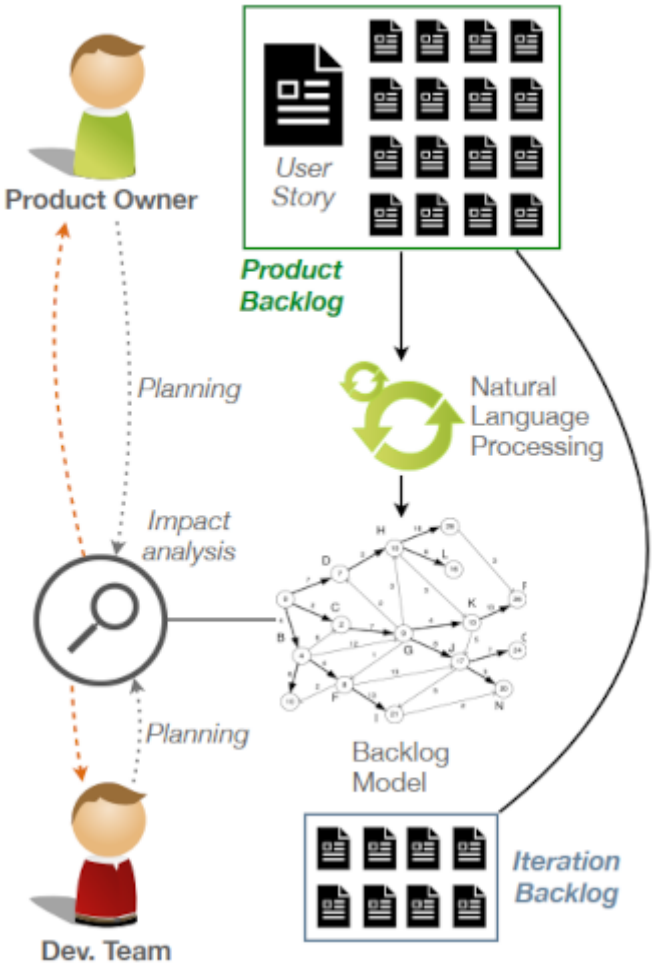

Figure 7: Providing early feedback at the backlog level [**?**]

Composable Backlogs

In order to support team customization (*e.g.*, a given team might want to enrich the backlog metamodel with additional information existing in their product management system) Mossser et al. chose open-world(ontological) representation by modelling backlog as graphs [**?**]. 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:*

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

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.

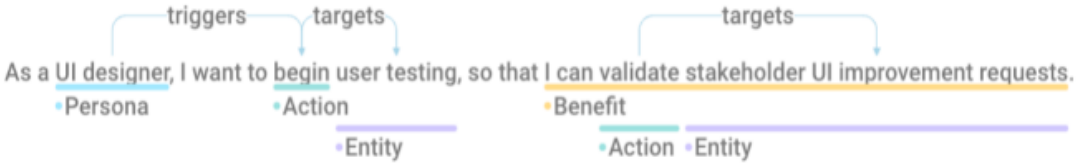

Figure 8: Example of annotated user using Doccano Annotation UI [**?**]

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 application 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

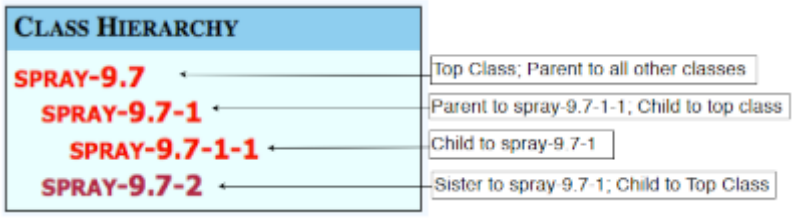

Figure 9: Class hierarchy for spray-9.7 class [**?**]

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.

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[**?**]

Conclusion

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

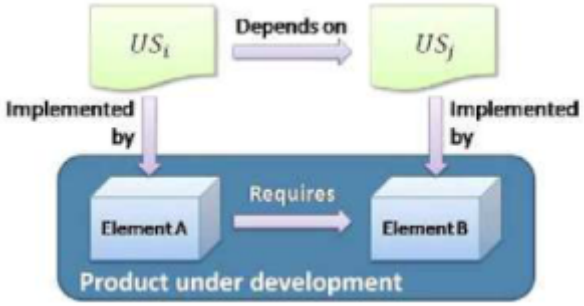

Figure 10: Inherited dependencies by user stories[**?**]

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, 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:*

[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_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."*

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

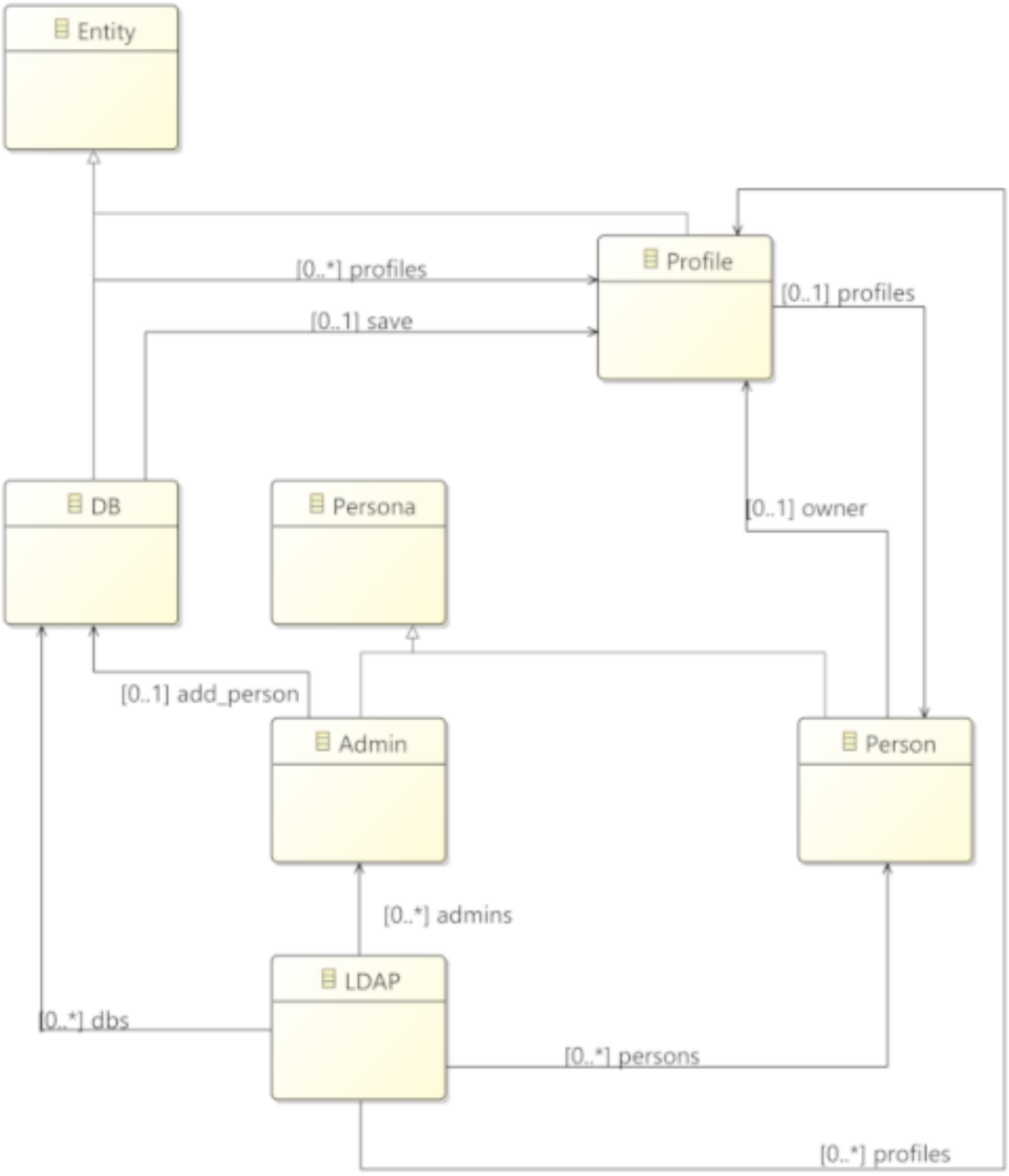

Figure 11: Henshin Class Model LDAP

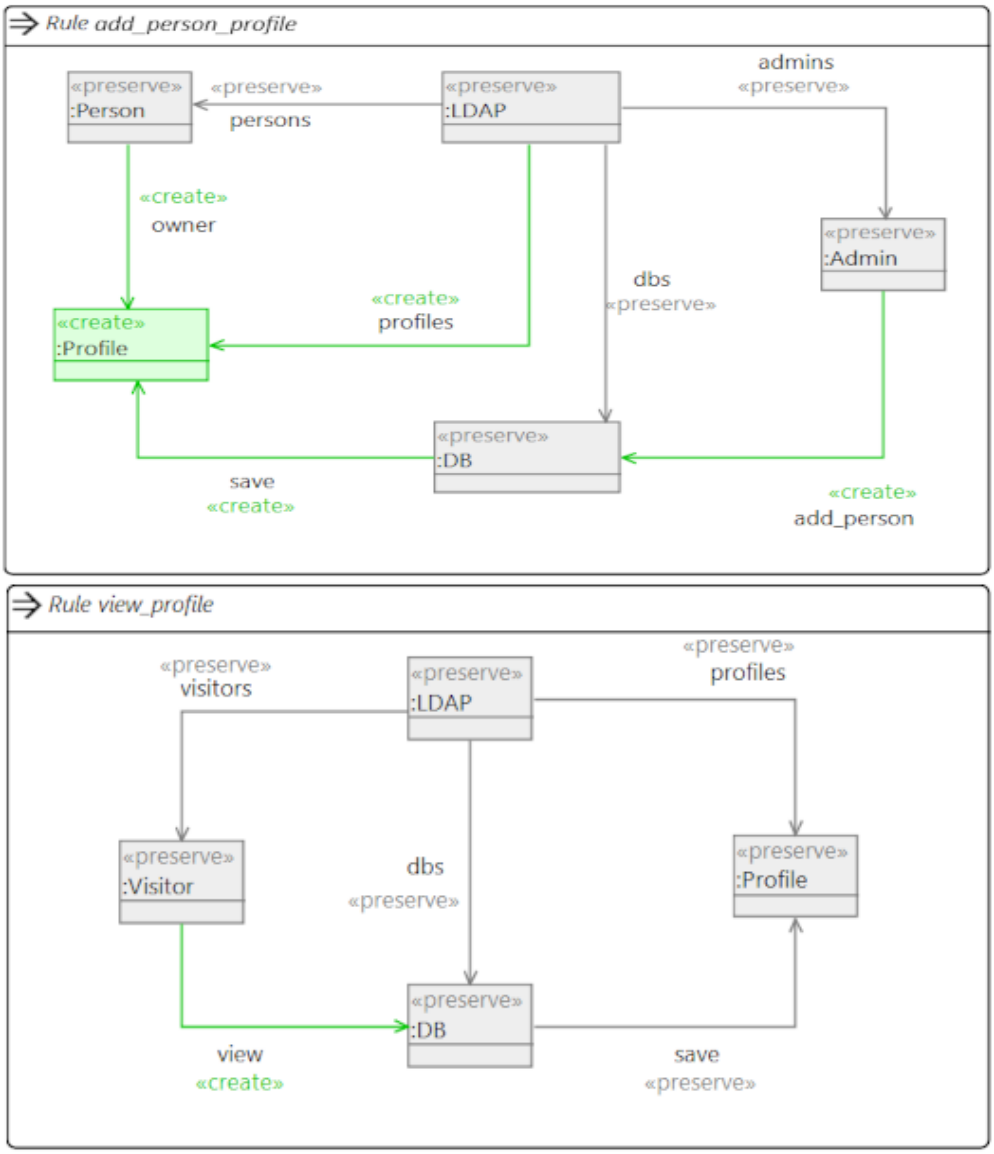

Figure 12: Illustrated rules in Henshin: view_profile rule related to US_2 and add_person related to US_1

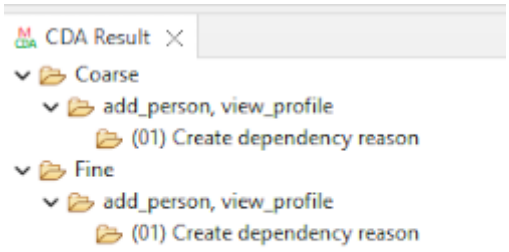

Figure 13: Henshin CDA result visualises dependencies between user stories

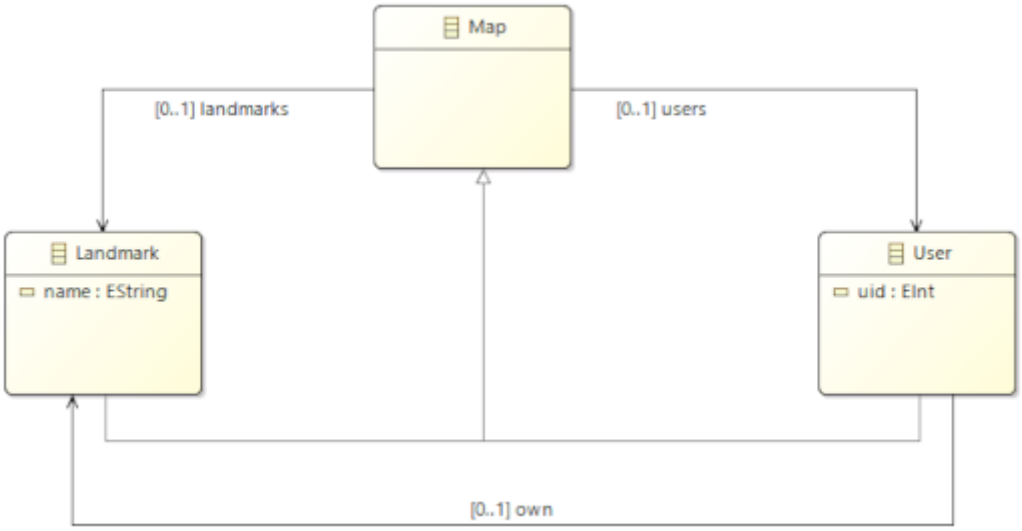

Figure 14: Henshin Class Model Map

user interface (UI) is provided supporting domain experts in developing rules by using the CPA interactively [**?**].

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.

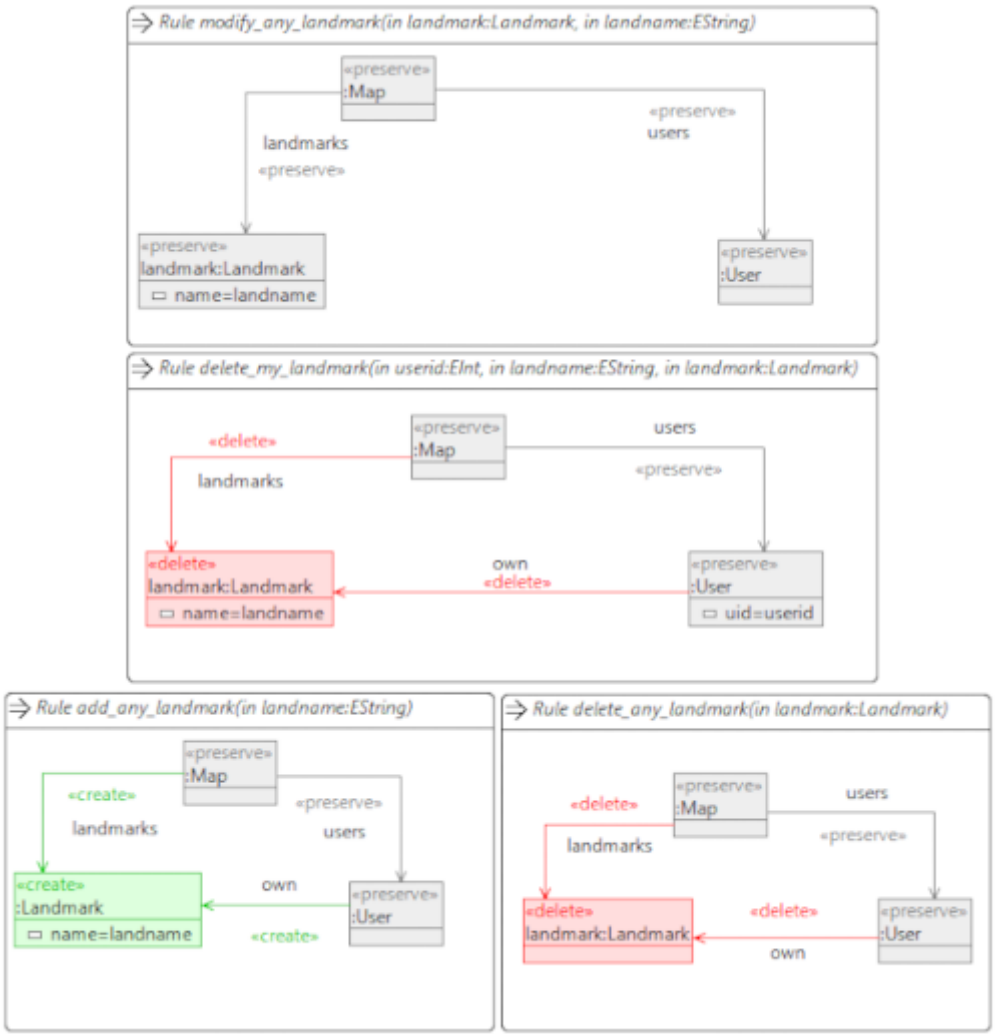

Figure 15: Illustrated rules in Henshin: add_any_landmark rule related to US_1a, modify_any_landmark rule related to US_1b, delete_any_landmark rule related to US_1c and delete_my_landmark rule related to US_2

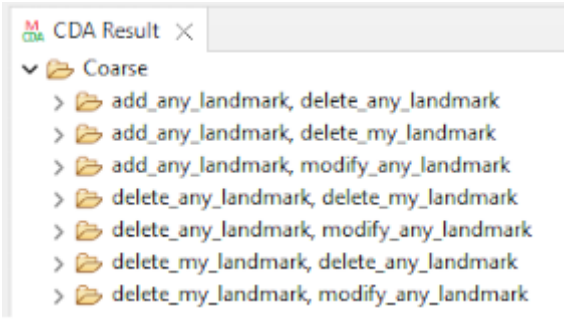

Figure 16: Henshin CPA result visualises conflicts and dependencies between user stories