Analysis of Conflicts and Dependencies between User Stories

Independent Scientific Work Computer Science

Amir Rabieyan Nejad

November 12, 2023

Abstract

In this scholarly paper, we expound upon an extensive assortment of techniques and tools that traverse diverse domains. These domains encompass agile software development, with a distinct emphasis on backlog management for the acknowledgment of user stories and their associated quality criteria. Furthermore, we delve into the realm of natural language processing (NLP) and its intricate interplay with computational lexicon resources, which is pivotal for tasks related to natural language comprehension. Additionally, we explore techniques of extracting domain models from textual requirements, which are instrumental in applying graph transformation rules and in discerning conflicts and dependencies among user stories.

The principal objective of our scholarly pursuit is to conduct a comparative analysis of these techniques and tools within each domain. Our intention is to harness this analysis for future endeavors, aiming to establish a meticulously structured workflow. This framework holds the promise of expediting the automated detection of conflicts and dependencies inherent within user stories expressed in natural language.

Contents

1 Introduction

User stories, the fundamental building blocks of software development, serve as concise and testable descriptions of a program's functionality. Within the dynamic framework of agile development, these user stories are typically composed informally, using plain text, and they are maintained in the product backlog, which acts as a repository for prioritizing and tracking development tasks.

Behavior Driven Development (BDD), a specialized approach within the realm of agile software development, places a strong emphasis on the iterative implementation of user stories. The sequencing of user stories in BDD is a pivotal aspect of the methodology. The right sequence not only impacts the efficiency of development but also the overall success of the project. By prioritizing and sequencing user stories effectively, development teams can deliver incremental value to users, respond to changing requirements, and ensure that the most critical functionality is addressed first.

User stories often exhibit dependencies on one another, leading to potential conflicts in which one user story necessitates the deletion of a component vital to the successful execution of another user story, or one user story may introduce an element that contravenes and thus prohibits the realization of another user story. To minimize the occurrence of conflicts, teams should systematically identify and document dependencies between user stories within their backlog. Agile methodologies, such as Scrum, promote cross-functional collaboration and daily stand-up meetings as mechanisms for promptly addressing and mitigating dependencies and conflicts. However, this approach can be time and resource-intensive. In instances where the backlog is extensive, the recognition of existing conflicts between user stories can become a complex endeavor.

To achieve the automation of conflict and dependency resolution within the scope of user stories associated with a single backlog, it is imperative that we establish a well-structured workflow. This workflow should encompass a collection of techniques and tools derived from various domains.

In this research paper, we embark on a comprehensive exploration of cutting-edge techniques and methodologies in the realm of natural language processing (NLP) and computational lexicon resources, a symbiotic relationship that holds pivotal significance for tasks pertaining to natural language comprehension. Furthermore, we embark on an exploration of techniques which extracting domain models from textual requirements, pivotal disciplines enabling the application of graph transformation rules and the discernment of conflicts and dependencies among user stories.

Section 2 introduces fundamental concepts, providing the necessary background information about *user story* and *backlog*. Additionally, we delve into the techniques employed in the User Story Quality and compare these techniques with each other. Moving forward to Section 3, we conduct a comparative analysis of various approaches extracting domain models from textual requirements with a specific focus on efficient backlog management geared towards the recognition of user stories. Section 4 is dedicated to a comparative analysis of several computational lexical resource techniques. The aim is to identify the most appropriate verb lexicon for the categorization of verbs found within user stories into three distinct categories, thereby facilitating the formulation of precise transformation rules. Section 5 centers on the comparison of methods for generating graph transformation rules. Finally, section 6 conclude the paper.

2 User Story Quality Analysis Techniques

In this section, we embark on an exposition of methodologies with the specific objective of refining the precision of User Stories measurement. First, in subsection 2.1 the role of USs and backlogs in agile development will be discussed. Additionally, the most common pattern of US will be introduced.

From subsection 2.2 our focus centers on the exploration of established techniques to address the query regarding the existence of criteria for the management and identification of conflicts and dependencies among User Stories. Furthermore, our attention is directed towards the criteria they employ, a pivotal factor in ensuring the creation of well-structured and standardized User Stories. Subsequent to this exploration, we undertake a comprehensive comparative analysis of these methodologies to ascertain the approach that best aligns with our particular contextual requirements.

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 [49] 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, incrementally. Each iteration selects a subset of the stories, maintaining a link between the developers and the end-users[40].

Ordinarily, User Stories typically exhibit interdependencies, wherein the order of their implementation becomes a critical consideration. This circumstance raises the question of how one may discern and identify the relationships that exist between the User Stories.

Sedano et al. posited that a “product backlog is an informal model of the work to be done” [49]. A backlog implements a shared mental model over practitioners working on a given product, acting as a boundary artifact 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 [1] Jira . These tools stores 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.

[1] Jira Software

A user story is a brief, semi-structured sentence and informal description of some aspect of a software system that illustrates requirements from the user's perspective [44]. 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 [55]. 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) [44].

The user story components consist of the following elements [52] : *Role*: abstract behavior of actors in the system context; the aspect of who representation. *Goal*: a condition or a circumstance desired by stakeholders or actors. Task: specific things that must be done to achieve goals. *Capability*: the ability of actors to achieve goals based on certain conditions and events.

2.2 Improving User Story using INVEST Criteria

In Agile software development, effective requirement management is crucial for delivering valuable and high-quality software. The INVEST criteria serve as a set of guiding principles that help teams assess and shape their user stories and requirements to meet the demands of Agile development. Each letter in the acronym INVEST represents a fundamental attribute that a requirement should possess to maximize its effectiveness within an Agile context.

These criteria were introduced to ensure that Agile teams create requirements that are not only clear and actionable but also adaptable to changing circumstances. The goal is to promote flexibility, collaboration, and a relentless focus on delivering value to the end-users or stakeholders [10].

In the following, we will examine the individual INVEST criteria in depth, with a focus on explaining their importance and their pragmatic application in the context of Agile Requirement Engineering [13].

Independent

Independent emphasize the avoidance of interdependencies between user stories or requirements. This principle underscores the importance of ensuring that each requirement operates autonomously, without relying on the completion of other related requirements. The presence of dependencies among requirements can introduce complexities in the prioritization and planning phases of Agile development. Such dependencies may necessitate a specific order of implementation, hindering the team's ability to adapt to changing priorities and potentially causing delays in project delivery.

Negotiable

USs (User Stories) should not be construed as rigid, contractual obligations or exhaustive lists of requirements that the software must adhere to. Instead, they are viewed as dynamic entities, open to ongoing discussion and refinement.

US are not meant to encapsulate every conceivable detail at the outset. The inclusion of excessive details can create a deceptive illusion of precision or completeness, potentially stifling the need for continued dialogue and collaboration.

An essential attribute of negotiable USs is their inherent flexibility. It is crucial to avoid treating User Stories as contractual obligations, as this can lead to a rigid mindset where estimates and commitments are made with the same level of inflexibility as a traditional contract. Instead, Agile teams should recognize the negotiable nature of USs and remain open to adapting them as circumstances warrant.

Valuable

The criterion of Value underscores that US must possess inherent worth and significance. A US should be designed in such a way that it delivers tangible value to either the end-users or the stakeholders. The value encapsulated within a US can manifest in various forms, such as improved user experiences, enhanced functionality, increased efficiency, or meeting specific business objectives.

Agile teams should prioritize and select USs that offer real, quantifiable value, as this ensures that the development effort is consistently directed toward outcomes that align with the project's overarching goals and objectives.

The concept of Value serves as a guiding principle that encourages Agile teams to continuously assess and prioritize USs based on their potential to contribute positively to the project's success, ultimately enhancing the overall quality and impact of the software being developed.

Estimable

An essential condition for a US to be deemed Estimable is that it must be sufficiently well-defined and clear. Ambiguity or lack of clarity within a US can render it inestimable, meaning that the development team cannot provide a reliable estimate of the effort required for implementation.

Estimable USs facilitate the planning process by enabling the team to make informed decisions about the allocation of resources, timeframes, and priorities. Clear and concise USs ensure that the estimation process is based on a solid foundation of understanding, leading to more accurate project planning and execution.

Small

The attribute of smallness is a pivotal aspect of the INVEST criteria for USs. It pertains to the size and granularity of individual USs. Large stories, present several challenges in Agile development. They tend to be intricate and challenging to estimate accurately.

A Complex Story refers to a US that is inherently large and resistant to disaggregation into smaller, more granular stories. Complex stories are a source of concern in Agile development because they can hinder effective planning and may require special consideration and strategies to address. Additionally, they pose difficulties in terms of fitting into single iterations or sprints.

Testable

The testable attribute signifies that USs should be structured in a manner that allows for the clear demonstration of whether they meet the customer's expectations. The verification of meeting these expectations is typically accomplished through testing.

An overarching goal in Agile development is to maximize the efficiency and effectiveness of testing processes. To achieve this, Agile teams strive for a high degree of test automation, with a target of attaining a test automation rate exceeding 90%.

Test automation plays a pivotal role in Agile development by enabling rapid and repeatable testing of software features. Automated tests ensure that the software remains consistent with the desired functionality outlined in the US, thereby reducing the risk of regressions and defects.

INVEST grid in table 1 can be used as a template when the customer and provider meet to evaluate a US [10].

The INVEST criteria serve as fundamental guidelines that are partially incorporated into the definition of many existing quality frameworks and requirement development tools in Agile methodologies.

INVEST	Description	0 Poor/Absent	1 Fair	2 Good	3 Excellent
I – Independent	User Stories should be as independent as possible	The start of construction of a US is tied to the completion of at least one other US	The completion of a US hinders the start of construction of at least one other US	The US can contain any constraint, but its release can be constrained by the completion of at least one other US	The US is fully independent, and it can be realized and released with any constraint
N – Negotiable	User Stories should be ”open”, reporting any relevant details as much as possible	The US contains enough detail to be a technical specification (Design phase), leaving no room to negotiate any element	The US is written with enough detail to be a functional specification (Analysis phase), leaving no room to negotiate any element	The US is written with informative content defining a User Requirement in a consolidated manner, yet shared between Customer and Provider	The US is written with the informative content typical of a high-level need, allowing feedback between customer and provider
V – Valuable	User Stories should provide value to end users in terms of the solution	The functional part (F) of the US does not contain all the functionalities requested by the customer	The functional (F) part of the US expresses mostly qualitative (Q) and technical (T) requirements about the system, and needs to be more developed in terms of functional requirements	The functional (F) part of the US expresses mostly the functional requirements requested by the Customer, but also includes qualitative (Q) and technical (T) requirements	The functional (F) part of the US correctly expresses only the functional requirements requested by the customer
E – Estimable	Each User Story must be able to be estimated in terms of relative size and effort	The US shows only its functional (F) part, filled in by the customer, but without sufficient detail to allow the provider to fill in the Q/T parts	The US shows only its functional (F) part, filled in by the customer, but validated with the provider	The US has been completed by the provider with respect to Q/T issues, but still needs to be validated jointly with the customer	All the useful parts of the US (F/Q/T) are shown, allowing the effort need to size and estimate it, and validated by both parts
S – Small	Each User Story should be sufficiently granular, and not defined at too high a level	The US is very large, and cannot be completed within a Sprint	The US is very large, and can be completed within a Sprint, but cannot accommodate the creation/delivery of other US	The size of the US is such that it can be completed within a Sprint, jointly with other US, but it is too small to create overhead about the Testing phase	The size of the US is such that it can be completed within a Sprint, jointly with other US, ensuring an appropriate balance between development and testing activities
T – Testable	Each User Story must be formulated in an effort to stress useful details for creating tests	The US does not include tips about Acceptance Tests	The US includes a formal indication of Acceptance Tests, but yet to be completed	The US includes an indication of Acceptance Tests which are complete, but yet to be validated	The US includes an indication of completed and validated Acceptance Tests

Table 1: INVEST Grid [10]

2.3 A Quality Framework

In this section we introduce the quality framework which use traditional requirement known as the Software Product Certification Model (SPCM)[23] to establish quality criteria for agile requirements. SPCM is based on extensive literature research for traditional up-front requirements engineering.

In order to certification two types of input are required: (1) one or more software artifacts and (2) on or more properties of these artefacts that are to be certified. The SPCM divides a software artifact into six Product Areas, namely the *Context Description*, which describes the environment of the system, the *User Requirement*, the *High-Level Design*, the *Detailed Design*, the *Implementation* and *Test*. Next to the division, the SPCM defines specific certification criteria for each area [23]. The properties of these artifacts which Heck et al. have denoted as "Conformance Properties", can fall into one of the following categories:

- Consistency: do the different (parts of) software artifacts conform to each other?
- Functional: does input to the system produce the expected output?
- Behavioral: does the system meet general safety and progress properties like absence of deadlocks or are constraints on the specific states of the system met?
- Quality: do the artifacts fulfill nonfunctional requirements in the areas of for example performance, security, and usability?
- Compliance: do the artifacts conform to standards, guidelines, or legislation?

Based on the SPCM they define three overall criteria for agile requirements [24]:
Completeness
All elements of the agile requirement should be present. Three levels have considered: *basic* elements, *required* elements, *optional* elements. In that way it can differentiate between elements that are absolutely mandatory for a requirement and elements that are nice to have because they increase the requirement quality.
Uniformity
The style and format of the agile requirements should be standardized, because this leads to less time for understanding and managing the requirements. Each time a team member is confronted with a new requirement he/she needs some time to understand the requirement and decide what to do with it. This process takes less time when the requirements format is standardized. Then all team members know where to look for what information on the requirement or how to read certain models attached to the requirement.
Conformance
Each element in the requirements is described in a correct and consistent way. The relations between the elements in the requirements description and with the context description are correct and consistent. They should be subject to manual verification, as: two requirements or use cases contradict each other; No requirement is ambiguous;

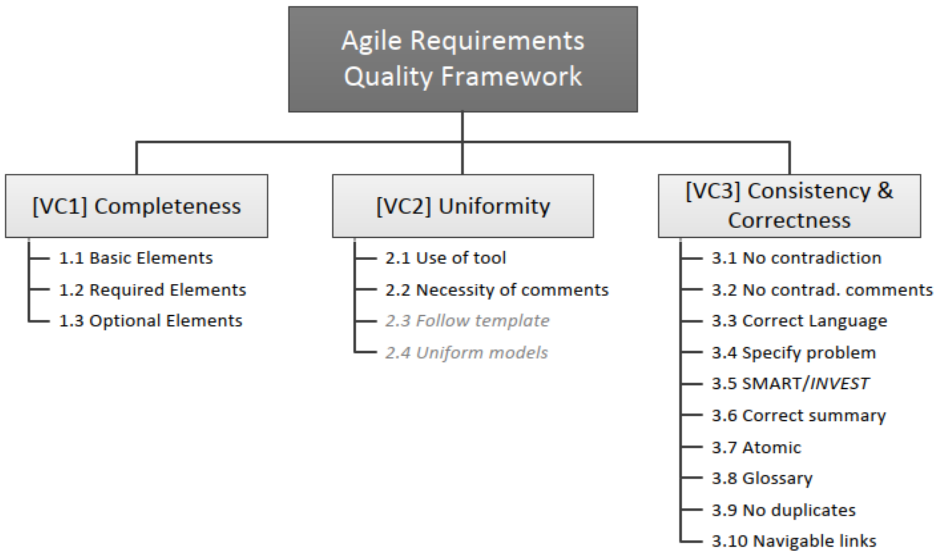


Figure 1: Agile Requirements Verification Framework [24]

Functional requirements specify what, not how; Each requirement is testable; Each requirement is uniquely identified; Each requirement is atomic; The glossary definitions are non-cyclic; Use case diagrams correspond to use case text; The data model diagram is in normal form.

The following criteria are explicitly delineated for USs, as depicted in figure 1:

- Basic Elements: Role, activity, business value ('Who needs what why?') instead of summary and description
- Required Elements: acceptance criteria or acceptance tests to verify the story instead of rationale
- Optional Elements: the team could agree to more detailed attachments to certain user stories (e.g. UML models) for higher quality
- Stories Uniform: each user story follows the standard user voice form
- Attachments Uniform: any modeling language used in the attachments is uniform and standardized

2.4 QUS framework

Lucassen et al. [34] represent a Quality User Story (QUS) framework, which consist of 13 quality criteria that US writers should strive to conform to. Subjected criteria determine the intrinsic quality of USs in terms of syntax, pragmatics, and semantics (Figure 2; Table 2). Base on QUS, Lucassen et al. present the Automatic Quality User

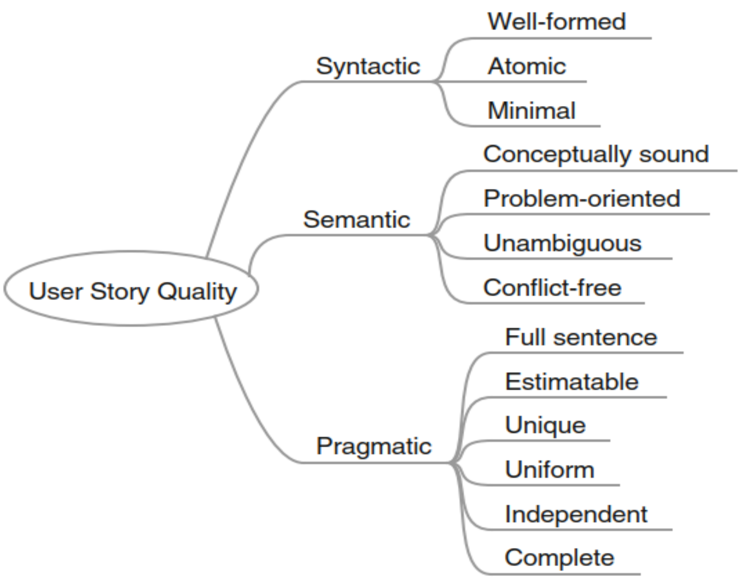


Figure 2: Agile Requirements Verification Framework [34]

Story Artisan (AQUSA) software tool to assessing and enhancing US quality. Relying on NLP techniques, AQUSA detects quality defects and suggest possible remedies.

A user story should follow some pre-defined, agreed upon template chosen from the many existing ones [53]. The skeleton of the template is called *format* in the conceptual model, in between which the *role*, *means*, and optional *end(s)* are interspersed to form a user story.

Because USs are a controlled language, the QUS framework's criteria are organize in Lindland's categories [30]:

- *Syntactic quality*, concerning the textual structure of a US without considering its meaning;
- *Semantic quality*, concerning the relations and meaning of (parts of) the US text;
- *Pragmatic quality*, considers the audience's subjective interpretation of the user story text aside from syntax and semantics.

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.

Independent

USs should not overlap in concept and should be schedulable and implementable in any order.

Complete independence may not always be achievable, the recommendation is to make any dependencies explicit and visible. Additionally, resolving certain dependencies

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

Table 2: Quality User Story framework that defines 13 criteria for user story quality: details [34]

may not be possible, and it is suggested practically approaches such as adding notes to story cards or using hyperlinks in issue trackers to make these dependencies evident. Two illustrative cases of dependencies are presented:

- *Causality*: In some cases, one user story (l_1) must be completed before another (l_2) can begin. This is formalized as the predicate "$hasDep(l_1, l_2)$", indicating that l_1 causally depends on l_2 when specific conditions are met.

- *Superclasses*: USs may involve an object (*e.g.*, "Content" in US "As a User, I am able to edit the content that I added to a person's profile page") that refers to multiple other objects in various stories, implying that the object in l_1 serves as a parent or superclass for the other objects.

Well-formed
Before it can be considered a US, the core text of the requirement needs to include a role and the expected functionality: the *means*. Considering the US "I want to see an error when I cannot see recommendations after I upload an article". It is likely that the US writer has forgotten to include the role. The story can be fixed by adding the role: "As a Member, I want to see an error when I cannot see recommendations after I upload an article.".
Atomic
A user story should concern only one feature. Although common in practice, merging multiple user stories into a larger, generic one diminishes the accuracy of effort estimation[31]. For instance, the US "As a User, I am able to click a particular location from the map and thereby perform a search of landmarks associated with that latitude longitude combination" consist of two separate requirements: the act of clicking on a location and the display of associated landmarks. This US should be split into two:

- US_A: "As a User, I'm able to click a particular location from the map";

- US_B: "as a User, I'm able to see landmarks associated with the latitude and longitude combination of a particular location".

Minimal
User stories should contain a role, a means, and (optimally) some ends. Any additional information such as comments, descriptions of the expected behavior, or testing hints should be left to additional notes. Consider 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": Aside from a role and means, it includes a reference to an undefined mockup and a note on how to approach the implementation. The requirements engineer should move both to separate user story attributes like the description or comments, and retain only the basic text of the story: "As a care professional, I want to see the registered hours of this week."
Conceptually sound
The means and end parts of a user story play a specific role. The means should capture a concrete feature, while the end expresses the rationale for that feature. Consider the

US "As a User, I want to open the interactive map, so that I can see the location of landmarks": The end is actually a dependency on another (hidden) functionality, which is required in order for the means to be realized, implying the existence of a landmark database which is not mentioned in any of the other stories. A significant additional feature that is erroneously represented as an end, but should be a means in a separate user story, for example:

- US_A: "As a User, I want to open the interactive map";
- US_B: "As a User, I want to see the location of landmarks on the interactive map.".

Problem-oriented
In line with the problem specification principle for RE proposed by Zave and Jackson [56], a user story should specify only the problem. If absolutely necessary, implementation hints can be included as comments or descriptions. Aside from breaking the minimal quality criteria, this US "As a care professional I want to save a reimbursement—add save button on top right (never grayed out)" includes implementation details (a solution) within the user story text. The story could be rewritten as follows: "As a care professional, I want to save a reimbursement.".
Unambiguous
Ambiguity is intrinsic to natural language requirements, but the requirements engineer writing user stories has to avoid it to the extent this is possible. Not only should a user story be internally unambiguous, but it should also be clear in relationship to all other user stories. The Taxonomy of Ambiguity Types [7] is a comprehensive overview of the kinds of ambiguity that can be encountered in a systematic requirements specification.

In this US "As a User, I am able to edit the content that I added to a person's profile page", "content" is a superclass referring to audio, video, and textual media uploaded to the profile page as specified in three other, separate user stories in the real-world user story set. The requirements engineer should explicitly mention which media are 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 added to a person's profile page.".
Full sentence
A user story should read like a full sentence, without typos or grammatical errors. For instance, the US "Server configuration" is not expressed as a full sentence (in addition to not complying with syntactic quality). By reformulating the feature as a full sentence user story, it will automatically specify what exactly needs to be configured. For example, US "Server configuration" can be modified to "As an Administrator, I want to configure the server's sudo-ers."
Estimatable
As user stories grow in size and complexity, it becomes more difficult to accurately estimate the required effort. Therefore, each user story should not become so large that estimating and planning it with reasonable certainty becomes impossible [2]. For example, the US "As a care professional I want to see my route list for next/future days, so that I

[2]INVEST in good stories, and SMART tasks

can prepare myself (for example I can see at what time I should start traveling)" requests a route list so that care professionals can prepare themselves.

While this might be just an unordered list of places to go to during a workday, it is just as likely that the feature includes ordering the routes algorithmically to minimize distance travelled and/or showing the route on a map. These many functionalities inhibit accurate estimation and call for splitting the user story into multiple user stories; for example:

- US_A: "As a Care Professional, I want to see my route list for next/future days, so that I can prepare myself";
- US_B: "As a Manager, I want to upload a route list for care professionals.".

The subsequent quality criteria pertain to a collection of USs. These quality criteria are instrumental in the assessment of the overall project specification's quality, focusing on the entirety of the project specification as opposed to the individual scrutiny of individual stories:

Unique and Conflict-Free

The concept of unique user stories, emphasizing the avoidance of semantic similarity or duplication within a project. For example considering EP_a: "as a Visitor, I am able to see a list of news items, so that I stay up to date" and US_a: " As a Visitor, I am able to see a list of news items, so that I stay up to date". This situation can be improved by providing more specific stories, like:

- US_{a1} "As a Visitor, I am able to see breaking news;"
- US_{a2} "As a Visitor, I am able to see sports news."

Additionally, the importance of avoiding conflicts between user stories should be considered to ensure their quality. ***A requirements conflict occurs when two or more requirements cause an inconsistency*** [42] [48]. For instance, considering 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 editing is a general term that includes deletion too, the conflict is "any landmark" versus "the landmark that I added". A possible way to fix this is to change US_c to: "As a User, I am able to edit the landmarks that I added." [34]

To detect these types of relationships, each US part needs to be compared with the parts of the other USs, using a combination of similarity measures that are either syntactic (*e.g.*, Levenshtein's distance) or semantic (*e.g.*, employing an ontology to determine synonyms). When similarity exceeds a certain threshold, a human analyst is required to examine the user stories for potential conflict and/or duplication.

Definition 2.1. *A user story μ is a 4-tuplel $\mu = (r, m, E, f)$ where r is the role, m is the means, $E = (e_1, e_2, ...)$ 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 is an action verb, do is a direct object, io is*

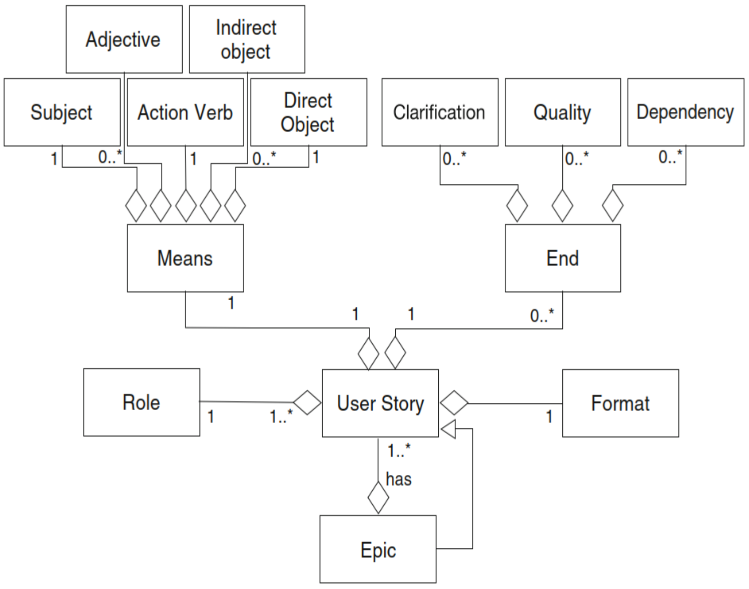


Figure 3: Conceptual model of user stories [34]

an indirect object, and adj is an adjective (io and adj may be null, see Figure 3). The set of user stories in a project is denoted by $U = (\mu_1, \mu_2, ...)$.

Definition 2.2. Different means, same end *Two or more user stories that have the same end, but achieve this using different means. This relationship potentially impacts two quality criteria, as it may indicate: (1) a feature variation that should be explicitly noted in the user story to maintain an unambiguous set of user stories, or (2) a conflict in how to achieve this end, meaning one of the user stories should be dropped to ensure conflict-free user stories. Formally, for user stories* μ_1 *and* μ_2*:*
$diffMeansSameEnd(\mu_1, \mu_2) \leftrightarrow m_1 \neq m_2 \wedge E_1 \cap E_2 \neq \emptyset$

Definition 2.3. Same means, different end *Two or more user stories that use the same means to reach different ends. This relationship could affect the qualities of user stories to be unique or independent of each other. If the ends are not conflicting, they could be combined into a single larger user story; otherwise, they are multiple viewpoints that should be resolved. Formally,*
$sameMeansDiffEnd(\mu_1, \mu_2) \leftrightarrow m_1 = m_2 \wedge (E_1 \setminus E_2 \neq \emptyset \vee E_2 \setminus E_1 \neq \emptyset)$

Definition 2.4. Full duplicate *A user story* μ_1 *is an exact duplicate of another user story* μ_2 *when the stories are identical. This impacts the unique quality criterion. Formally,*
$isFullDuplicate(\mu_1, \mu_2) \leftrightarrow \mu_1 =_{syn} \mu_2$

Definition 2.5. Semantic duplicate *A user story* μ_1 *that duplicates the request of* μ_2*, while using a different text; this has an impact on the unique quality criterion. Formally,*
$isSemDuplicate(\mu_1, \mu_2) \leftrightarrow \mu_1 = \mu_2 \wedge \mu_1 \neq_{syn} \mu_2$

Uniform
Uniformity pertains to the consistency of a USs format with the majority of user stories within the same set. To evaluate uniformity, the requirements engineer identifies the most frequently occurring format, usually established in collaboration with the team. For example, the US "As an Administrator, I receive an email notification when a new user is registered" is presented as a non-uniform user story and can be rewritten for improved uniformity as: "As an Administrator, I want to receive an email notification when a new user is registered."
Complete
The implementation of a set of USs should result in a feature-complete application. While it's not necessary for USs to cover 100% of the application's functionality up-front, it's crucial not to overlook essential USs, as doing so may create a significant feature gap that hinders progress. For instance, consider the US "As a User, I am able to edit the content that I added to a person's profile page", which requires the existence of another story describing content creation. This scenario can be extended to USs with action verbs that reference non-existent direct objects, such as reading, updating, or deleting an item, which necessitates its creation first. To address these dependencies related to the means' direct object, Lucassen et al. introduce a conceptual relationship.

2.5 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 user story practice [34]. 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 [34]. AQUSA consists of five main architectural components (Figure 4): linguistic parser, US base, analyzer, enhancer, and report generator.

The first step for every US is validating that it is well-formed. This takes place in the linguistic parser, which separates the user story 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 [34].

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.

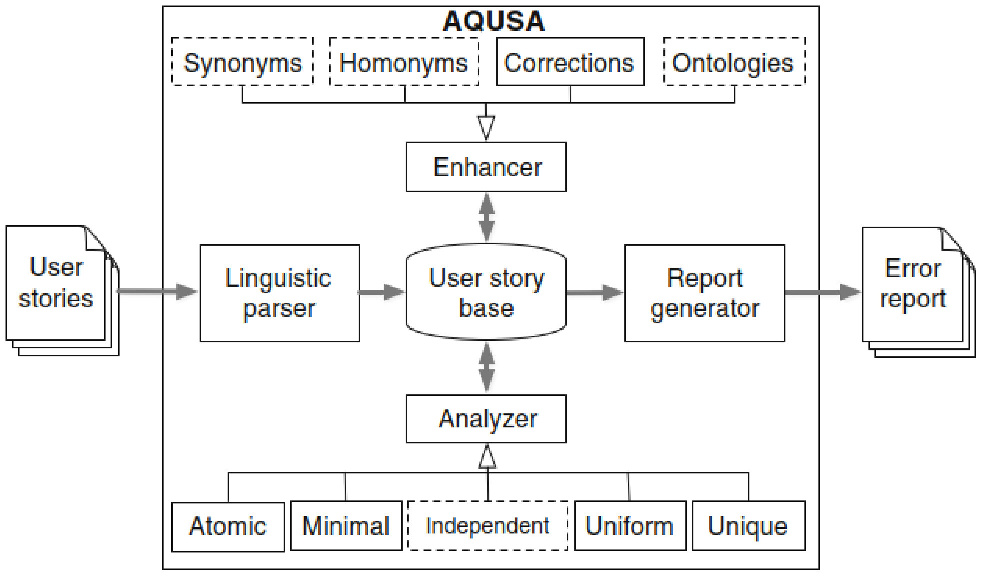


Figure 4: Functional view on architecture of AQUSA. Dashed components are not fully implemented yet [34]

Linguistic Parser: Well-Formed
One of the essential aspects of verifying whether a string of text is a user story 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 user story, 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 user story 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 user story 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 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

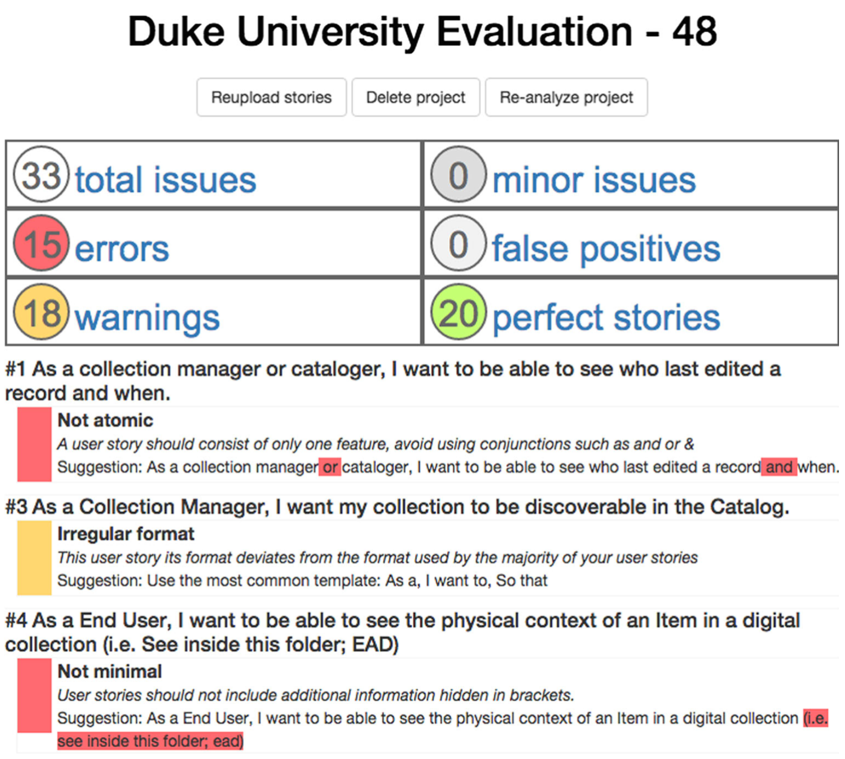
AQUSA has the capability to utilize various similarity measures, leveraging the WordNet lexical database to detect semantic similarity. For each verb and object found in the means or end of a US, AQUSA performs a WordNet::Similarity calculation with the corresponding verbs or objects from all other USs. 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.

2.6 Comparative Analysis

In this subsection we consider a comparative analysis of mentioned methodologies to discern the most suitable approach for our specific context. The IEEE Recommended Practice for Software Requirements Specifications defines requirements quality on the basis of eight characteristics [15]: correct, unambiguous, complete, consistent, ranked for importance/stability, verifiable, modifiable, and traceable. The standard, however, is generic and it is well known that specifications are hardly able to meet those criteria



Duke University Evaluation - 48

33 total issues	0 minor issues
15 errors	0 false positives
18 warnings	20 perfect stories

#1 As a collection manager or cataloger, I want to be able to see who last edited a record and when.

Not atomic
A user story should consist of only one feature, avoid using conjunctions such as and or &
Suggestion: As a collection manager or cataloger, I want to be able to see who last edited a record and when.

#3 As a Collection Manager, I want my collection to be discoverable in the Catalog.

Irregular format
This user story its format deviates from the format used by the majority of your user stories
Suggestion: Use the most common template: As a, I want to, So that

#4 As a End User, I want to be able to see the physical context of an Item in a digital collection (i.e. See inside this folder; EAD)

Not minimal
User stories should not include additional information hidden in brackets.
Suggestion: As a End User, I want to be able to see the physical context of an Item in a digital collection (i.e. see inside this folder; ead)

Figure 5: Example report of a defect and warning for a story in AQUSA [34]

[17].

With agile requirements in mind, the Agile Requirements Verification Framework [24] defines three high-level verification criteria: completeness, uniformity, and consistency and correctness. The framework proposes specific criteria to be able to apply the quality framework to both feature requests and USs. Many of these criteria, however, require supplementary, unstructured information that is not captured in the primary US text.

Quality frameworks often lack dedicated tools for tasks such as verifying unique IDs or establishing relationships between elements, making it necessary to rely on external tools like Jira for such checks. Other sides the QUS Framework focuses on the intrinsic quality of the US text. Other approaches complement QUS by focusing on different notions of quality in RE quality such as performance with USs [33] or broader requirements management concerns such as effort estimation and additional information sources such as descriptions or comments [24]. AQUSA, in conjunction with the QUS framework, is the only existed tools which offers several compelling reasons to consider its use for managing and enhancing USs in our approach:

- **Conflict and Dependency between US:** The INVEST criteria's independence attribute effectively manages conflicts and dependencies among USs, ensuring that US do not overlap conceptually, allowing for flexible scheduling and implementation.

  Conversely, in a Quality framework, conflict and dependency between USs are regarded as conformance verification criteria, with a focus on ensuring that no two requirements or use cases are contradictory and that each requirement has a unique identifier. It's worth noting that the Quality framework does not encompass the verification of unique IDs or the establishment of relationships between elements; instead, third-party software is typically tasked with handling this.

  Within the Quality User Stories (QUS) framework, independence is emphasized as a pragmatic quality criterion, requiring that US be self-contained and devoid of inherent dependencies on other stories. Analyzing independencies through AQUSA v1 is a component of Lucassen et al. forthcoming efforts. They emphasis that is improbable that it will fully meet the Perfect Recall Condition unless a significant breakthrough in the computer's comprehension of NLP emerges. This implies that the automated approach for analyzing conflicts and dependencies remains unresolved.

- **Enhanced Quality Assurance:** AQUSA and the QUS framework provide a systematic approach to ensure the quality of USs. They help identify and rectify various defects, such as missing information, inconsistencies, and ambiguities, leading to more robust requirements.

- **Defect Detection:** AQUSA automates the detection of defects within user stories, including well-formedness issues, uniqueness violations, minimal story violations, and more. This helps reduce the chances of miscommunication and misinterpretation among development teams.

- **Efficient Workflow:** AQUSA streamlines the process of verifying USs, making it easier for requirements engineers and developers to focus on resolving issues rather than manually inspecting every story.
- **Perfect Recall Condition:** AQUSA aims to fulfill the Perfect Recall Condition, minimizing the need for manual verification of USs. This can significantly reduce the risk of missed defects.
- **Quality Assurance Across Tools:** AQUSA's ability to detect and highlight issues, adds an extra layer of quality assurance, helping ensure that USs in popular issue tracking tools like Jira and Pivotal Tracker are well-structured and coherent.
- **Adaptability:** AQUSA can be customized to suit specific requirements and integrate with various issue tracking systems, making it adaptable to different development environments.
- **Independence and Uniqueness:** AQUSA helps maintain the independence and uniqueness of USs, which is crucial for scheduling and implementing stories in any order without causing conflicts or overlaps.

2.7 Conclusion

A closer look at the INVEST regarding dependencies and conflicts between USs, reveals that INVEST places great emphasis on promoting the autonomy of USs in order to dispense with their active management, which is mainly due to the complicated nature of such a management during the planning and prioritization phase. This perspective promotes a scenario in which USs can progress independently of the completion of other USs. Although this approach is undeniably idealistic, the pragmatic reality often deviates from this ideal. In practice, USs are often interdependent, so such linkages are almost inevitable.

With regard to the quality framework, it can be seen that there are no explicit criteria relating to conflicts and dependencies between USs, but that it follows the INVEST criteria.

Moreover, it is worth noting that neither framework comes with built-in tools for the automatic analysis of USs. The Quality Framework for User Stories (QUS) has a tool called AQUSA that automates the reporting of discrepancies in USs with respect to the QUS criteria.

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 Extracting Domain Models form Textual Requirements

Automated support for extracting domain models from requirements artefacts 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 artefacts and the whole system.

In this section, we present a comprehensive exploration of techniques that can be employed for the extraction of domain models from agile product backlogs. Furthermore, we conduct a comparative analysis of these techniques.

3.1 Visual Narrator: (Automated) Extraction of Conceptual Models form US

Visual Narrator automatically derive conceptual modes from a concise and widely adopted NL notation for USs. Robeer et al. combine NLP heuristics into an algorithm that creates conceptual modes from USs. They present the Visual Narrator as a fully automated, opensource tool that implements the algorithm and generates conceptual models as Web Ontology Languages (OWL) [47].

A Generic Conceptual Model

Robeer et al. define a generic conceptual model of stories that dissects a single US and define its syntactic structure [35].

The conceptual model is assembled as a UML class diagram in figure 6. US follow a standard predefined format [54] to capture three distinct aspects of a requirement:

- Who wants the functionality;
- What functionality the end users or stakeholders want the system to provide; and
- Why the end users and stakeholders need this functionality (optional).

These three aspects are captured in a simple textual template to form a running sentence. Although many different templates exist, 70% of practitioners use the template "As a <type of user>, I want <some goal>[so that <come reason>]" [19]. The conceptual model distinguishes between the *role*, *means* and *ends* parts of a US. Each of these parts features an *indicator* which delimits the three basic parts of a US. Jointly, the indicators define the *template* of a user story.

The role part encompasses the role indicator and the *functional* role, describing a generalized type of person *who* interacts with the system. When no ambiguity exists, Robeer et al. use the term *role* to denote both the full part and the *functional role*. The *means* consists of a *subject*, a main *object* of the functionality, and a *main verb* describing the relationship between the subject and main object. The main object can be represented explicitly by the direct object or implicitly (*e.g.*, "I want to log in" actually, refers to logging in the *system*).

Robeer et al. identified three main purposes that the *ends* may take: (1) clarify the means, (2) reference another user story, or (3) introduce a qualitative requirement.

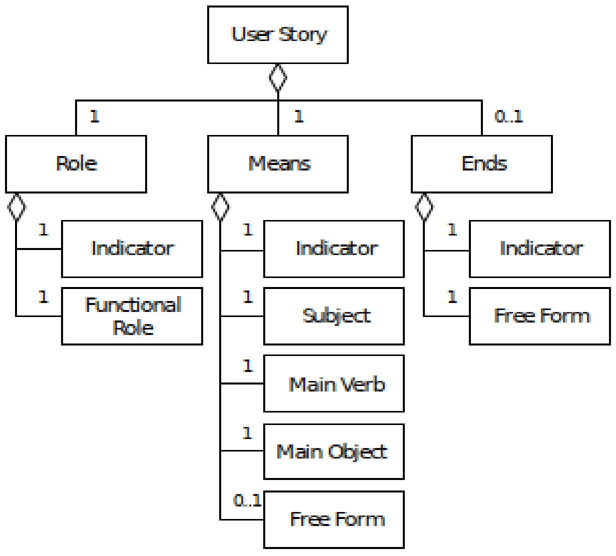


Figure 6: Conceptual model defining the syntax of a US [47]

The Visual Narrator Tool

To enable the automated extraction of conceptual modes from USs, Robeer et al. developed the *Visual Narrator* tool on the basis of the 11 heuristics [47] which takes a set of USs as input and generates a conceptual model as output. The tool only accepts USs that use the indicator as identified by Wautelet [54]: As — *As a(n)* for the role, *I want (to)* — *I can* / *I am able* / *I would like* for the means, and *so that* for the ends part. Syntactically invalid USs are not processed; in order to sanitize these stories, analysts should pre-process them using tools such as AQUSA [34].

The architecture of Visual Narrator comprises two main components: (1) the *Processor* and the (2) *Constructor*. The Processor analyzes USs, parsing them into tokens using spaCy [3] and determining token weights based on frequency and user-defined parameters. This results in a *Term-by-User-Story* matrix with weighted terms.

The Constructor then generates a conceptual model from the *WeightedTokens*. It involves the *PatternIdentifier*, which applies heuristics to identify patterns in USs, and the *PatternFactory*, which creates an internal conceptual model based on these patterns and filters out concepts and relationships with weights below a user-specified *threshold*. The Ontology component stores this model, linking parts of it to their originating USs.

To extract a conceptual model from USs, Visual Narrator implements the procedure DERIVECM which takes as input a set of raw user stories S and empty sets of concepts C and relationships R, and populates C and R while parsing the stories and applying the 11 heuristics.

[3] *https://spacy.io/*

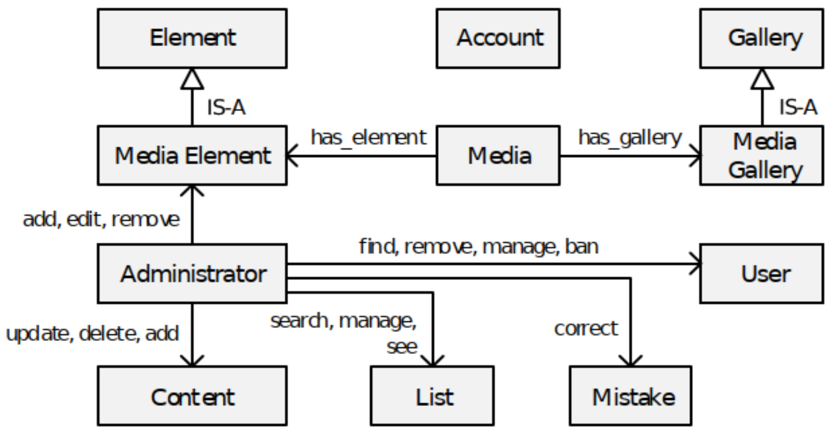


Figure 7: Partial model for WebCompany generated with Visual Narrator [47]

Example 3.1. *The WebCompany is a young Dutch company that creates tailor-made web business applications. The team consists of nine employees who iteratively develop applications in weekly Scrum sprints. WebCompany supplied 98 user stories covering the development of an entire web application focused on interactive story telling that was created in 2014. 73 of these 98 USs were syntactically correct, usable and relevant for conceptual model generation [34]. Part of the generated conceptual model is shown in figure 7.*

3.2 A Modelling Backlogs as Composable Graphs

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

The tool can extract what is called a conceptual model of a backlog in an ontology-like way. The conceptual models are then used to measure USs quality by detecting ambiguities or defects in a given story [40]. From a modelling point of view, Mosser et al. represents the concepts involved in the definition of a backlog in a metamodel, as depicted in figure 8. 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).

Consider, for example, the following story, extracted from the reference dataset [14]: "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

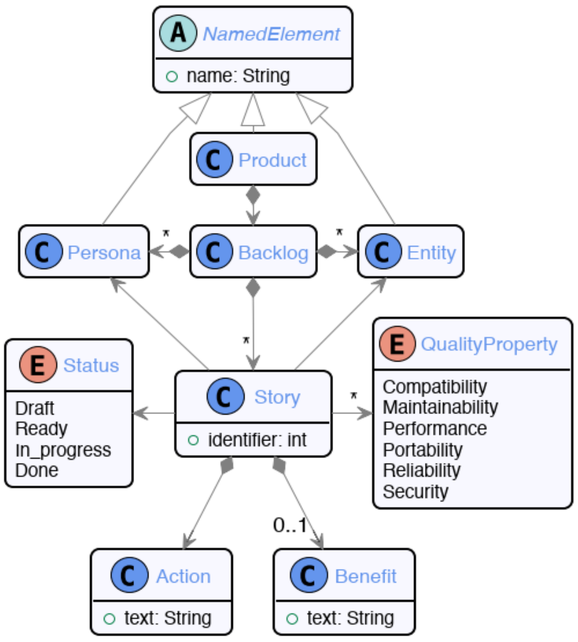


Figure 8: Backlog conceptual metamodel [40]

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 9 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 [40].

Composable Backlogs

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

Definition 3.1 (**Story**). *A 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*

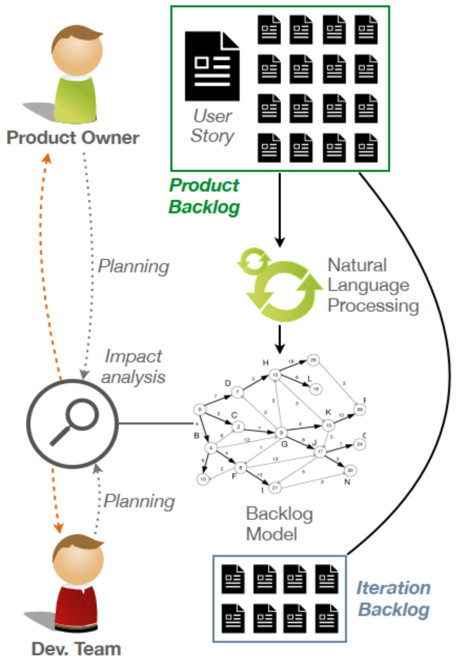


Figure 9: Providing early feedback at the backlog level [40]

$S : \forall (s, s') \in S^2, s \neq s' \Rightarrow s \not\equiv s'$.

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 3.2. *Backlog excerpt: Content Management System for Cornell University — CulRepo* [14].

- *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").

$b_1 = (V_1, E_1, \emptyset) \in B$
$V_1 = \{Persona(faculty\ member, \emptyset)$,

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

Conditional Random Fields (CRF)

CRFs [28] 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 [5].

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 [5]. 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) [5].

Example 3.3. *Consider the following example:*

$$
\begin{aligned}
&S = ['As', 'a', 'UI', 'designer', ',', \ldots] \\
&POS(S) = [ADP, DET, NOUN, NOUN, PUNCT, \ldots] \\
&Label\,(S) = [\emptyset, \emptyset, PERSONA, PERSONA, \emptyset, \ldots]
\end{aligned}
$$

S represents a given US (Table 3). $POS\,(S)$ 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 3.

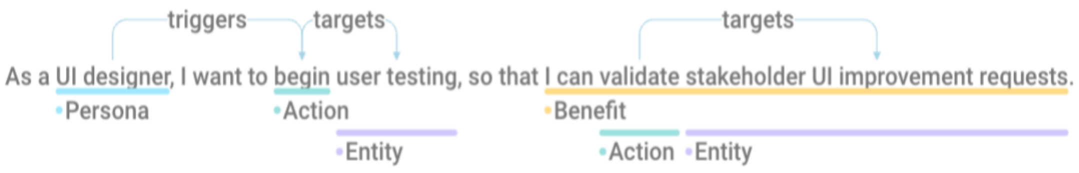

triggers
targets
targets
As a UI designer, I want to begin user testing, so that I can validate stakeholder UI improvement requests.
Persona
Action
Benefit
Entity
Action
Entity

Figure 10: Example of annotated user using Doccano Annotation UI [5]

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 3: Minimal Feature Set, associating part-of-speech (POS) and semantic labels to each word in a given story [5]

Extracting Domain Models with GPT 3.5

Mooser et al. experimented with ChatGPT to find the best way to extract the information from a US backlog in a *"rapid prototyping"* way. They initially used a batch approach to extract information from stories using a GPT agent. However, they faced limitations due to token limits and hallucinations. To address this, they switched to a one-by-one story approach, reducing hallucinations but still producing unusable results.

They then explored function calls introduced in *gpt-3.5-turbo-0613* [4], allowing control over output which make developer able to provide a JSON schema in order to model their response, and the system will use this schema and *"fill in the blanks"* instead of regular text generation.

Example 3.4. *If one expects their answer to be an array of strings containing the name of the personas, they can provide a schema inside their request, and GPT will use it as output format (as in Figure 11, line* 3 → 12*). An example of such a constrained response is described in Figure 12.*

To address limitations in GPT's output format control, the Mosser et al. introduced the option to use JSON schemas in requests, allowing users to specify the desired output format. They also explored the use of function calls as responses but found it impractical due to conversation termination. Faced with this, they had two options: defining a large schema or engaging in a conversation with the model. They chose the latter, designing the processing of each story as a conversation with a smaller, dedicated schema.

[4] https://platform.openai.com/docs/guides/gpt/completions-api

```
response = openai.ChatCompletion.create(
    model = "gpt-3.5-turbo-0613",
    functions = [ #constraining_GPT_with_a_schema
        { "name": "record_elements", #Function to be called in the
            ↪ response
          "description":
              ↪ "Record_the_elements_extracted_from_a_story",
          "parameters": { #Signature_description
                "type": "object",
                "properties": {
                "personas": {
                        "type": "array",
                        "description": "The list of personas
                            ↪ extracted from the story",
                        "items": { "type": "string" }}}}],
    messages = conv,
    temperature=0.0)
        ↪ #To_make_the_answer_deterministic_(as_much_as_possible)
```

Figure 11: Calling GPT 3.5 and specifying a function call argument [5]

```
{
    ...
    "choices": [{
        "index": 0, "message": {
         "role": "assistant", "content": null,
         "function_call": {
            "name": "record_elements",
                    "arguments":
                 "{\"personas\": [\"repository manager\"]}"
    }},
 "finish_reason": "function_call"}],
    ...
}
```

Figure 12: Example of answer from the API (execution of Figure 11) [5]

Eventually, they adopted a conversation-based approach with smaller JSON schemas to process each story effectively. They organized the conversation into four phases:

1. Setup. First, the system role will be impersonated and ask the engine to adopt a persona.

2. Concepts. The task of extracting personas, entities, actions and benefit from a story will be proceed.

3. Categorization. Given stateless of the model, it becomes necessary to inject the answer obtained from the preceding phase into the ongoing conversation. Consequently, Mooser et al. assume the role of the assistant and include a conversation entry detailing the <CONCEPTS> obtained in the previous phase. Following the established pattern from the prior phase, Mooser et al. articulate the task using the system role, which entails categorizing primary and secondary actions and entities.

4. Relations. The final step uses the same pattern. Mosser et al. first inject the <CATEGORIES> as the assistant, and then describe the task and provide an example as the system.

3.3 Comparative Analysis

In this subsection we consider a comparative analysis of mentioned methodologies to discern the most suitable approach for our specific context.

In the context of evaluating three approaches (Visual Narrator, GPT-3.5, and Conditional Random Fields or CRF) for the task of domain concept extraction from a corpus of stories, it is evident that CRF emerges as a compelling choice. Here are the reasons why CRF should be chosen for our approach:

- Tailored Approach: Mosser et al. highlights CRF as a rule-based approach, allows for a tailored and domain-specific model. This tailored approach is crucial when dealing with complex domain-specific tasks like concept extraction. CRF can be fine-tuned by domain experts to address the specific challenges of the task.

- Training Requirement: CRF requires training, it was trained using a classical 80/20 separation method. This training phase allows CRF to learn the patterns and relationships between domain concepts and the textual context. This training can lead to improved accuracy and relevance in domain concept extraction (Figure 13) [5].

- Model Complexity: In comparison to other approaches, CRF's implementation is relatively simple, and it requires a smaller codebase. This simplicity makes it easier to manage and maintain, reducing the complexity associated with more extensive models like GPT-3.5.

- $F - measure$ (F_1) : Mosser et al. measures performance using the F_1 score, which is particularly suitable for this task. F_1 considers both precision and recall, which is essential when dealing with domain concept extraction. It is well-suited for situations where false positives and false negatives have different costs [5].

- Superior Performance: Empirical results indicate that CRF consistently outperformed other approaches, including GPT-3.5, in all evaluation criteria. CRF achieved the highest F_1 scores, particularly for Persona's extraction, reaching a perfect score of 1.0 which means a perfect match (score of 0.0 means that precision or recall is null). This superior performance is indicative of CRF's effectiveness in this specific task [5].

- Reproducibility: CRF offers transparency and interpretability, making it easier to understand and reproduce results compared to more opaque machine learning models.

- Concerning the Agile DevOps Paradigms: Visual Narrator keeps its conceptual model internal and exposes black-box analysis to users, dedicated to stories quality in terms of requirements engineering, for example. Thus, it does not support the DevOps team in the development, as the provided feedback focuses on the requirements expression instead of their role in the software development. CRF instead leverage the graph structure of the ontologies and exploit a graph-based meta-modelling and compositional approach to shorten the feedback loop between developers and Ops while supporting agile development's iterative and incremental nature [40].

- Syntactic and semantic Covering: In contrast to GPT-3.5 and CRF, which consider both the syntactic and semantic aspects of US, NLP has constrained its focus solely to the syntactic structure of the US.

3.4 Conclusion

The evaluation in the paper [5] concludes already that CRF is better than the alternatives. Additionally, we have chosen the Conditional Random Fields (CRF) approach because of their graph-based nature and their significant promise in terms 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 an appropriate conceptual metamodel, which makes it 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, improving support for DevOps practices.

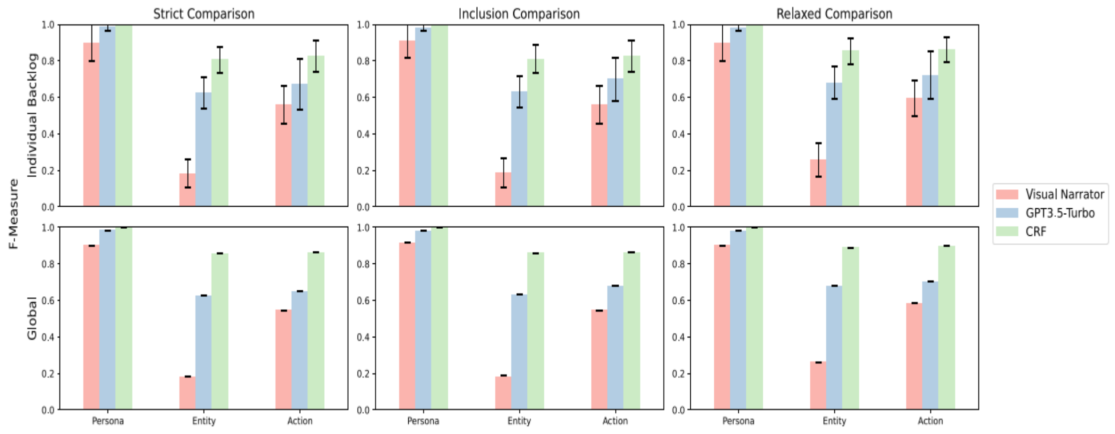


Figure 13: Comparing approaches to the ground truth: $F - measure$ (F_1) results for Visual Narrator, CRF and GPT-3.5 [5]

4 NLP and Computational Lexical Resource Techniques

Natural language processing (NLP) is a computational method for the automated analysis and representation of human language [11]. 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 [4], extraction of actors and actions in requirement document [1].

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 [11].

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 behavior of words, have attracted considerable interest in NLP [11]. 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 behavior of relevant words [26].

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” 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” and “forbiddance” of linguistic elements embedded within the construct of the US.

Next, we will undertake a systematic examination of various computational lexical resource techniques. Subsequently, we will conduct a comparative analysis to determine the optimal approach for integration into our approach.

4.1 WordNet

WordNet is an online lexical database designed for use under program control. English nouns, verbs, adjectives, and adverbs are organized into sets of synonyms, each representing a lexicalized concept. Semantic relations link the synonym sets [38].

WordNet operates by classifying words into four core syntactic categories: nouns, verbs, adjectives, and adverbs, collectively referred to as open-class words (see Table 4). This categorization allows for versatile language analysis and interpretation. For instance, words like "back," "right," or "well" can be interpreted differently based on their linguistic context, and thus, WordNet separately incorporates these varied interpretations [37].

WordNet not only addresses word forms but also considers their inflectional morphology, accommodating the linguistic nuances that can arise due to different syntactic categories. For instance, when information is sought for “went”, the system intelligently provides relevant details about the verb “go”. On the other hand, derivational and compound morphology, encompassing words like “interpret”, “interpreter”, “misinterpret”, and so forth, are acknowledged as distinct entities within WordNet, despite sharing a common root.

One of WordNet’s key strengths lies in its incorporation of various semantic relations, meticulously chosen to serve the broader scope of the English language. These relations are designed to be user-friendly and intuitively understandable, making them accessible even to individuals without advanced linguistic training. WordNet’s semantic relations encompass [37]:

- Synonymy: The fundamental relation in WordNet, which employs sets of synonyms (synsets) to encapsulate word senses. This relation is symmetric and connects different word forms that share the same sense.

- Antonymy (opposing-name): A symmetric relation that highlights opposing word forms, particularly vital in organizing the meanings of adjectives and adverbs.
- Hyponymy (sub-name) and Hypernymy (super-name): These transitive relations between synsets organize noun meanings into hierarchical structures, with hypernyms representing broader categories and hyponyms representing specific instances or subcategories.
- Meronymy (part-name) and Holonymy (whole-name): These complex relations delineate the component parts, substantive parts, and member parts of various concepts within WordNet.
- Troponymy (manner-name): Similar to hyponymy but applicable to verbs, troponymy results in hierarchies that are typically shallower.
- Entailment Relations: WordNet also accounts for entailment relations between verbs, capturing the logical inferences that can be drawn from certain actions.

These semantic relations are represented in WordNet through pointers, establishing connections between word forms and synsets. In total, WordNet incorporates more than 116,000 of these pointers, facilitating the exploration of intricate semantic relationships.

4.2 FrameNet

The Berkeley FrameNet project is dedicated to creating machine-readable semantic descriptions for a multitude of English words. The project's primary goal is to encode human semantic knowledge into machine-readable formats, guided by empirical findings from corpus-based research. The project ambitiously covers various semantic domains, such as HEALTH CARE, CHANCE, PERCEPTION, COMMUNICATION, TRANSACTION, TIME, SPACE, BODY, MOTION, LIFE STAGES, SOCIAL CONTEXT, EMOTION, and COGNITION. The Berkeley FrameNet project plays a significant role in advancing natural language understanding and semantic analysis [6].

The Berkeley FrameNet project yields two crucial outcomes: the FrameNet database and associated software tools. The database comprises three major components:

- Lexicon: This section includes entries containing conventional dictionary-type data for human readers' comprehension. It also incorporates FORMULAS, which capture the morphosyntactic structures within phrases or sentences built around words. Links to semantically ANNOTATED EXAMPLE SENTENCES illustrate these realization patterns found in the formulas. Additionally, there are connections to the FRAME DATABASE and other machine-readable resources like WordNet and COMLEX.
- Frame Database: This component provides descriptions of each frame's basic conceptual structure, offering names and descriptions for the elements participating in these structures. Table 5 schematizes several related entries in this database [6].

Semantic Relation	Syntactic Category	Example
Synonymy (similar)	N, V, Aj, Av	pipe , tube rise, ascend sad, unhappy rapidly, speedily
Antonymy (opposite)	Aj, Av, (N, V)	wet, dry powerful, powerless friendly, unfriendly rapidly, slowly
Hyponymy (subordinate)	N	sugar maple, maple maple, tree tree, plant
Meronymy (part)	N	brim, hat gin, martini ship, fleet
Troponomy(manner)	V	march, walk whisper, speak
Entailment	V	drive, ride divorce, marry

Note: N = Nouns Aj = Adjectives V = Verbs Av = Adverbs

Table 4: Semantic Relation in WordNet [37]

frame (TRANSPORTATION)
frame.elements (MOVER(S), MEANS, PATH)
scene (MOVER(S) move along PATH by MEANS)
frame (DRIVING)
inherit (TRANSPORTATION)
frame.elements (DRIVER (=MOVER), VEHICLE)
(=MEANS), RIDER(S) (=MOVER(S)), CARGO)
(=MOVER(s)))
scenes (DRIVER starts VEHICLE, DRIVER controls
VEHICLE, DRIVER stops VEHICLE)
frame (*RIDING*$_1$)
inherit (TRANSPORTATION)
frame_elements (RIDER(S) (=MOVER(S)), VEHICLE
(=MEANS))
scenes (RIDER enters VEHICLE, VEHICLE carries
RIDER along PATH, RIDER leaves VEHICLE)

Table 5: A subframe can inherit elements and semantic from its parent [6]

- Annotated Example Sentences: These sentences are marked up to showcase both semantic and morphosyntactic properties of lexical items. These sentences lend empirical support to the lexicographic analyses found in the frame database and lexicon entries.

These three components are tightly integrated, with elements in each able to reference elements in the other two. Moreover, the database will include estimates of the relative frequency of senses and complementation patterns by comparing the senses and patterns in hand-tagged examples with the entire BNC corpus [6].

4.3 VerbNet

VerbNet (VN) is a hierarchical domain-independent, broad-coverage verb lexicon with mappings to several widely-used verb resources, including WordNet [37], Xtag [43], and FrameNet [6]. It includes syntactic and semantic information for classes of English verbs derived from Levin's classification which is considerably more detailed than that included in the original classification.

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

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 behavior, 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 [29], and class numbers 58-109, developed later by Korhonen and Briscoe [27]. 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

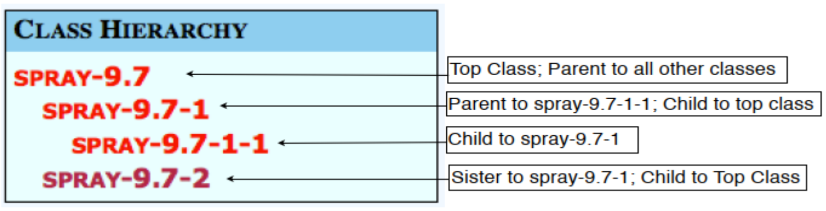

CLASS HIERARCHY
SPRAY-9.7
SPRAY-9.7-1
SPRAY-9.7-1-1
SPRAY-9.7-2
Top Class; Parent to all other classes
Parent to spray-9.7-1-1; Child to top class
Child to spray-9.7-1
Sister to spray-9.7-1; Child to Top Class

Figure 14: Class hierarchy for spray-9.7 class [24]

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 behaviors 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 behavior 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 amongst 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 14 illustrate an example of class hierarchy form spray-9.7 class. Verb Classes are 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).

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 6: An example of top-class numbers and their corresponding verb-type[41]

An example of top-class numbers and their corresponding types is given in Table 6. 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.

4.4 Comparative Analysis

After a careful comparative analysis of WordNet, FrameNet, and VerbNet, we have determined that VerbNet is the most suitable lexical resource for our project. Its hierarchical categorization of verbs into classes provides a structured and comprehensive approach to understanding verb behavior, which is essential for our goal of generating transformation rules based on semantic interpretations of actions within user stories.

VerbNet's organization aligns seamlessly with our project's requirements, offering the precision and granularity required for our semantic analysis and rule generation tasks.

Here are the three comparison key aspects:

- Focus: VerbNet is specialized in verbs, FrameNet offers broader coverage, including nouns and adjectives, and WordNet covers a wide range of words across different parts of speech.
- Hierarchy: VerbNet and FrameNet provide hierarchical structures that help organize and understand word behavior, while WordNet focuses on synsets and relationships between words.
- Applications: VerbNet is ideal for tasks related to verb semantics and actions. FrameNet is versatile for various lexical semantic tasks, and WordNet is widely used for word sense disambiguation and related applications.

4.5 Conclusion

For our approach we identified VerbNet as the most appropriate lexical resource. Its methodical categorisation of verbs into hierarchical classes provides a structured and all-encompassing framework for understanding verb behavior. This alignment with our project goals is of utmost importance as it underpins our task of formulating transformation rules based on semantic interpretations of the actions (verbs) recognised by CRF and described in USs. VerbNet's organizational structure seamlessly matches the requirements of our project and ensures the necessary precision and granularity that are essential for our semantic analysis and rule generation.

5 Analysis by Graph Transformation Tools

In this section, some basic definitions about graph transformation and transformation rules are first established for better understanding. Afterwards, we dive into graph transformation tools, which play a pivotal role in our methodology.

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 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.

Graphs and Typed Graphs

A graph is comprised 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 5.1 (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 5.2 (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 5.3 (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, 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 three distinct categories, namely: "create", "delete", "forbid". These categories serve as essential components of transformation rules that articulate precise changes within the graph-based representation.

In the following subsections, we will introduce two prominent graph transformation tools, Groove and Henshin, shedding light on their respective capabilities and applicability within our CRF-driven metamodeling framework for US annotation and transformation. Finally, we illustrate comparative analysis of the most suitable tool for our specific use case.

5.1 GROOVE: Modelling and analysis Tool

Graphs in groove consist of labelled nodes and edges. An edge is a binary arrow between two nodes. Node labels can either be node types or flags; the latter can be used to model a Boolean condition, which is true for a node if the flag is there and false if it is absent.

GROOVE can work either in an untyped or in a typed mode. In the untyped mode, graphs can be arbitrary: there are no constraints on the allowed combinations of node types, flags and edges. In the typed mode, all graphs and rules must be well-typed, meaning that they can be mapped into a special type graph. This is checked statically for the start graph and the rules: the theory then ensures that well-typedness

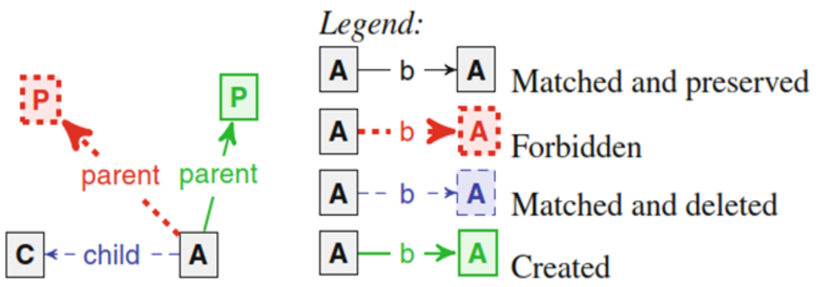


Figure 15: Example GROOVE rule and legend [16]

is preserved under transformation. The type graph determines the allowed combinations of node types and edges [16].

Rules and Application Condition

Graphs are transformed by applying rules. A rule consists of the following:

- A pattern that must be present in the host graph in order for the rule to be applicable
- Subpatterns that must be absent in the host graph in order for the rule to be applicable
- Elements (nodes and edges) to be deleted from the graph
- Elements (nodes and edges) to be added to the graph
- Pairs of nodes that are to be merged

All these elements are combined into a single graph; colors and shapes are used to distinguish them. Ghamarian et al. distinguish positive (which must be present in order to apply a rule) and negative (which must be absent in order to apply a rule) ones, whereas of the latter, Ghamarian et al. distinguish deletion and creation of elements. Figure 15 shows a small example illustrating most of these concepts:

- The black (continuous thin) “reader” elements, in this case two nodes labelled **A** and **C**, must be present and are preserved—in fact, they form a positive application condition.
- The red (dashed fat) “embargo” elements, in this case a **parent**-labelled edge with a **P**-labelled target node, must be absent in the graph—in fact, each connected subgraph of embargo elements forms a negative application condition.
- The blue (dashed thin) “eraser” elements, in this case a **child**-labelled edge from the **A**-node to the **C**-node, must be present and are deleted.
- The green (continuous fat) “creator” elements, in this case a parent-labelled edge with a **P**-labelled target node, are created.

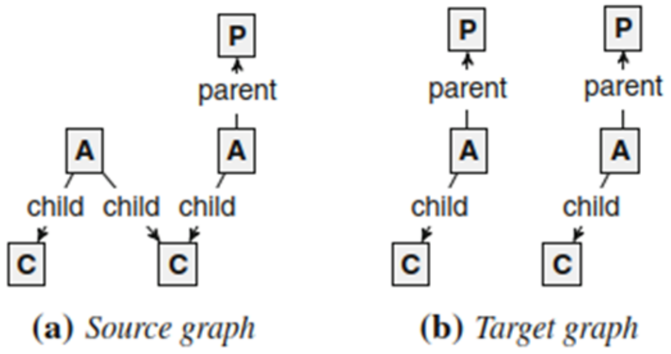


Figure 16: Example application of the rule in Figure 15

The overall effect of the rule is to search for ***A***- and ***C***-nodes connected by a ***child***-edge but without a ***parent***-edge to a ***P***-node, and to modify this by removing the ***child***-edge and adding a ***parent***-edge to a fresh ***P***-node.

For instance, the rule can be applied to the graph on the left-hand side of Figure 16 in two ways, one of which result in the graph on the right-hand side. (The other application removes the other ***child***-edge.)

The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph transformation system) to a given start graph, and to all graphs generated by such applications. This results in a state space consisting of the generated graphs. The strategy according to which the state space is explored (*e.g.*, depth-first, breadth-first or linear) can be set as a parameter [16].

Attributes

Nodes in a graph typically stand for instances of some resource or concept. It is also necessary to include data fields, containing Booleans, integer numbers or strings. Such data fields are usually called *attributes*.

GROOVE supports attributes by treating them as special edges that do not point to a standard node, but to a node that corresponds to a data value. Graphically, such edges are usually represented by expressions of the form “x = 12”, rather than by x-labelled arrows pointing to a $12 - labelled$ node.

Regular expressions

Besides ordinary edges, a rule may include edges carrying regular expressions. These will be matched in the host graph by searching for a path whose labels satisfy the regular expression.

This especially allows the specification of cycles or the transitive closure of edges. Regular expressions may also contain *wildcards*, which are matched by any label in a given set. Moreover, wildcard may be *named*; such a name is effectively a variable for edge labels.

Quantification

One of the special features of GROOVE is the support of universal quantification in rules [46]. A universally quantified (sub)rule is one that will be applied to all subgraphs that satisfy the relevant application conditions, rather than just a single one as in the standard case.

Such a rule can itself be much more concise, and also result in a smaller state space, then the equivalent set of rules that would ordinarily be needed. In fact, quantification can be nested in the sense that universally quantified rules can contain further existential subrules, and vice versa. Among other things, this makes it possible to formulate powerful application conditions [45].

Control

The standard behavior of GROOVE is to attempt the application of arbitrary rules at any point in time. There are, however, two further methods to control and direct the application of rules. A most straightforward mechanism is to assign *priorities* to rules: low-priority rules may only be applied if no higher-priority rule is applicable. A more sophisticated mechanism is to use GROOVE's *control language*. A control program is imposed on top of a graph transformation system and specifies the allowed order of application of the rules of that system, referring to the rules by name.

For instance, the control program a; try {b;} else {c;} specifies that first the rule named "a" must be applied, after which "b" is tried; if "b" is not applicable, "c" is applied. If rule "a" is not applicable in the beginning, then nothing happens. Other constructs offered by the language include:

- Looping, including an "as-long-as-possible" construct
- A random choice between rules
- Simple (noun-recursive) function calls

State space exploration

The most distinguishing feature of GROOVE, compared with another graph transformation tools, is the fact that it does not just carry out a single sequence of transformations from a given start state, but can explore and store the entire state space of reachable graphs. This provides a rich source of information for further analysis. In fact, GROOVE offers a choice of the exploration strategy to be used:

- Depth-first full exploration, also with on-the-fly Linear Temporal Logic model checking
- Breadth-first full exploration. In some grammars, this enables finding shortest paths to certain graphs
- Linear, random linear, and conditional exploration. This allows simulation without covering all states, for instance if the state space is too large.

In [34] Kleppe et al. describe the execution semantics of a simple object-oriented programming language in terms of graph transformation rules. A program graph is used as input and each rule application simulate the execution of a program instruction. By means of GROOVE's state space exploration capabilities, it is possible to generate finite execution traces of a program and model check for errors. In this setting, GROOVE can be seen as a non-deterministic execution engine for the language defined by the transformation rules.

GROOVE can provide a great assistance in analyzing US, where non-determinism as well as parallelism is an essential part. However, the problem does not scale in GROOVE for problems with large sizes. This is because the size of the state space grows dramatically as the ring size increases. This is the well-known state space explosion problem, common to all model checking tools [45].

Example 5.1. *To exemplify model checking in GROOVE, we consider a finite transition system consisting of two USs, US_1:"As an Administrator, I am able to add a new person to the database" and US_2: "As a Visitor, I am able to view a person's profile". The application of typed graph and transformation rules, as depicted in Figure 17, utilizing two transformation rules: one corresponds to US_1 named add_person; it serves the purpose of saving a Profile in the database when there is no existing Profile (corresponding to specific Person in the database). Another rule corresponding to US_2 named $view\_profile$ is responsible for adding a "view" edge between Visitor and the DB nodes if a Profile (corresponding to the Person) already exists in the database. AS we can see both rules are typed according to defined type graph .As start graph we choose the existent of four Element namely Admin, DB, Visitor and Profile.*

The application of GROOVE model checking to analyze transformation rules and a finite transition system is illustrated in Figure 18. As we can see, GROOVE found only a single sequence of transformation, which means, US_2 cannot proceed without execution and consideration of US_1 due to lack of **save**-edge between **Profile**-node and **DB**-node.

5.2 Henshin: A Tools for In-Place EMF Model Transformations

The Eclipse Modeling Framework (EMF) provides modeling 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 [2].

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 [50].

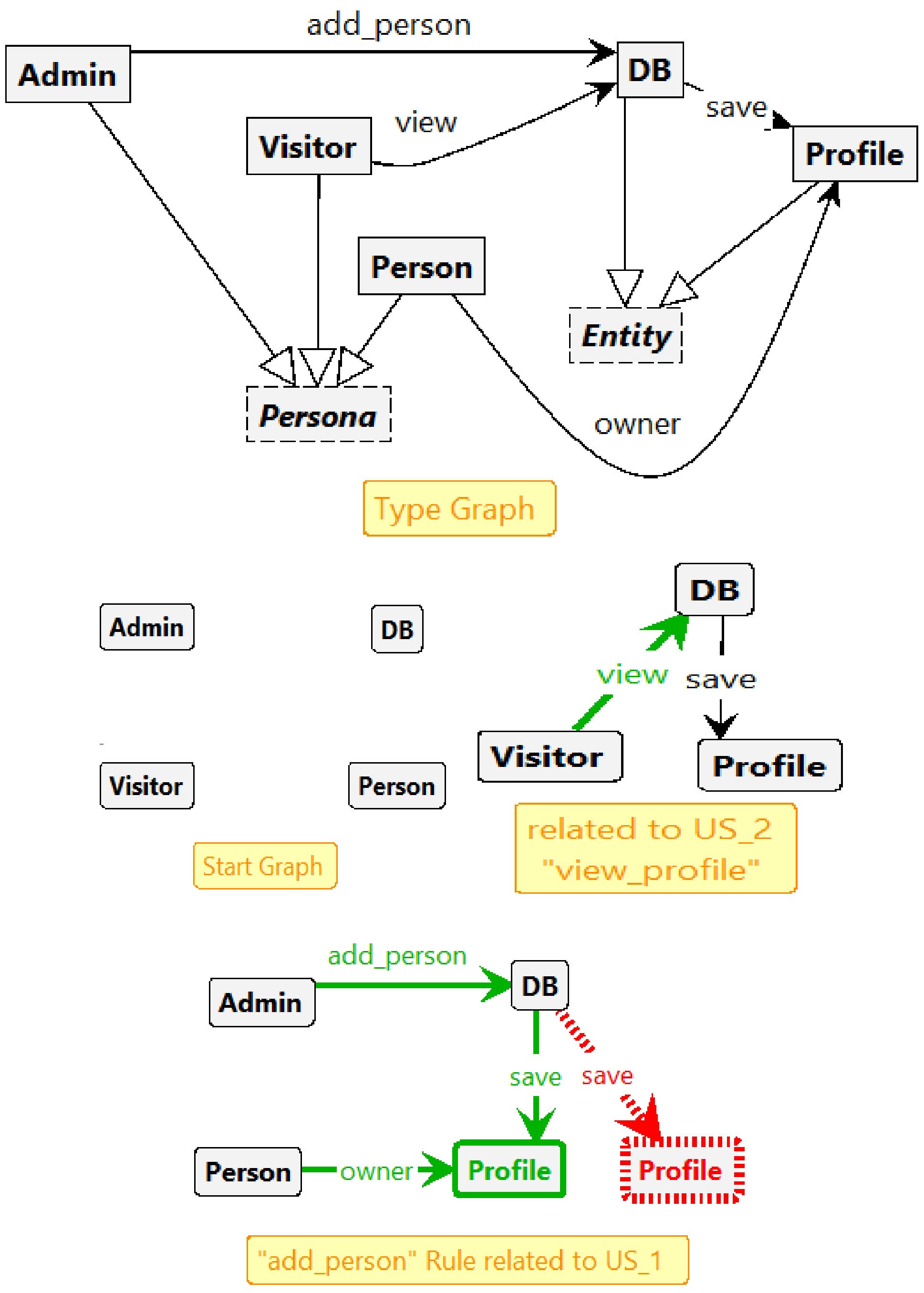


Figure 17: Provided Type-Graph, Start Graph and Transformation Rules for US_1 and US_2

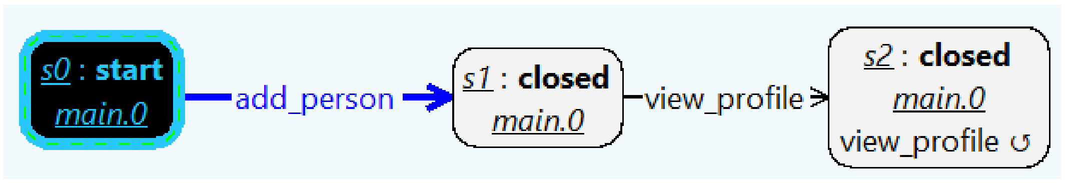


Figure 18: Application of GROOVE model checking to Analyze Transformation Rules and a finite transition system for two USs

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 [50]. To be efficient, graph transformation tools usually employ heuristics such as search plans to provide good performance (e.g. [51]).

Structure and Application of Rules

The Henshin transformation language is defined by means of a meta model. The Henshin meta model is closely aligned to the underlying formal model of double pushout (DPO) graph transformations [50]. 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 meta model 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 [2].

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 [20] 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 simulate all possible executions of a transformation for a given input model, and to apply model checking, similar to the GROOVE [25] tool.

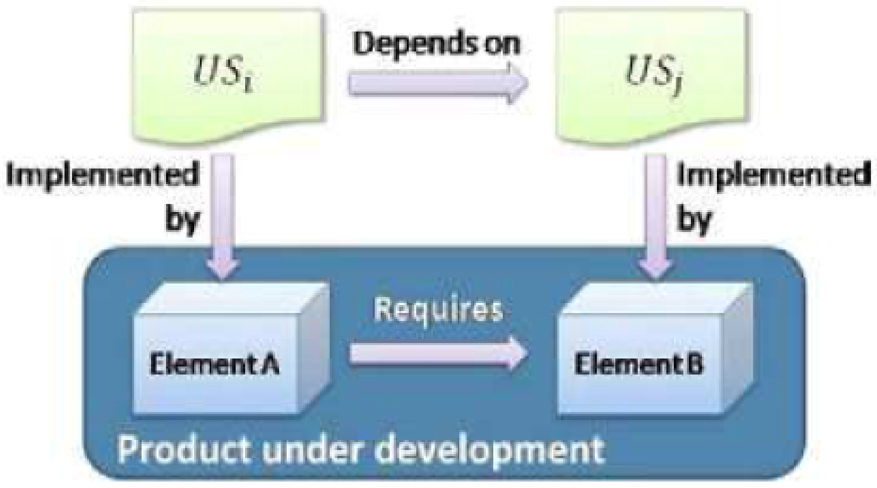


Figure 19: Inherited dependencies by user stories[18]

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 [25]. In Figure 19, *element A* requires *element B*, generating a dependency between them. Such dependencies are naturally inherited by the user stories (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 [12]. The existence of dependencies between USs makes necessary to have some implemented before others [12],[19],[32]. If the order of USs implementation does not consider these dependencies it may have a large number of preventable refactoring, 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 [18]. The critical pair analysis (CPA) for graph rewriting [21] can be adapted to rule-based model transformation, *e.g.*, to find conflicting functional requirements for software systems [22], or to analyze potential causal dependencies between model refactoring [36] which helps to make informed decisions on the most suitable refactoring to apply next. The CPA reports two different forms of potential causal dependencies, called conflicts and dependencies [8]. 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

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 7: Comparative analysis between model checking and conflict and dependency methods

– 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)**[5].

The subsequent dependency results differ in their target of the second attribute movement. The first produce/use-dependency (2) represents the case of moving the attribute back to the original class, which leads to a smaller minimal model with only two classes referencing each other, as depicted in Figure 25. The highlighting by enclosing hash marks is the most important information, since the enclosing element is the cause of the dependency. The link between 2:Class and 3:Attributeis created by the first rule application and is required by the second application. Since all elements and values in the minimal model may be matched by the first and the second rule application, there is a generic approach to represent attribute values. Value r1_source_r2_target , *e.g.*, means that it must conform to value source in rule r_1 and value target in rule r_2 , respectively (compare Figure 23(a)). The second dependency reported in Figure 24 is the handling of two consecutive attribute shifts [36].

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 7, which serves to elucidate their appropriateness for modeling USs.

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

EMF Refactoring Model

Since refactoring is a specific kind of model transformation, refactoring of EMF-based models can be specified in Henshin and then integrated into a refactoring framework such as EMF refactor [3].

Example 5.2. *Similar to the presented instance in GROOVE (as demonstrated in example 5.1), we exemplify conflicts and dependencies within Henshin using identical user stories (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 20 delineates the class model LDAP (Lightweight Directory Access Protocol). In Figure 21, the defined rules in Henshin, specifically the rule view_profile linked to US_2, and add_person corresponding to US_1, are depicted. The representation uses black to signify object preservation and green for new objects. Additionally, Figure 22 (CDA result) showcases the dependencies among the USs, illustrating that in Henshin, a specialized instance graph is not requisite.*

Example 5.3. *To demonstrate the main idea of refactoring in Henshin Born et al. represent an example for class modeling limited to one rule [9]. Rule* Move_Attribute *(Figure 23 (a)) specifies the shift of an attribute from its owning class to an associated one along a reference. It is shown in abstract syntax. Objects and references tagged by* `<<preserve>>` *represent unchanged model elements, elements tagged by* `<<create>>` *represent new ones whereas those tagged by* `<<delete>>` *are removed by the transformation [36].*

Modifying the class model in Figure 23(b) by the refactoring specified in Figure 23(a), Born et al. observe two potential problems: (1) the attribute landlineNo *of class Person can be shifted to either class Home or class* Office *(by refactoring* Move_Attribute*) [36]. However, if it is shifted to class Home the other refactoring becomes inapplicable (and vice versa). This means, refactoring* Move_Attribute *is in conflict with itself. (2) The attribute street of class* Person *can be shifted to class* Address *via class* Home *(by two applications of* Move_Attribute *along existing references). The second shift is currently not possible since class* Home *does not have an attribute so far, i.e., refactoring Move_Attribute may depend on itself. Graph transformation theory allows us to analyze such conflicts and dependencies at specification time by relying on the idea of the CPA [36].*

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 [36].

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.

Figure 24 depicts an example for the analysis of rule *Move_Attribute*, in which the *delete/use conflict* (1) corresponds to the example illustrated in figure 25 [36]. The

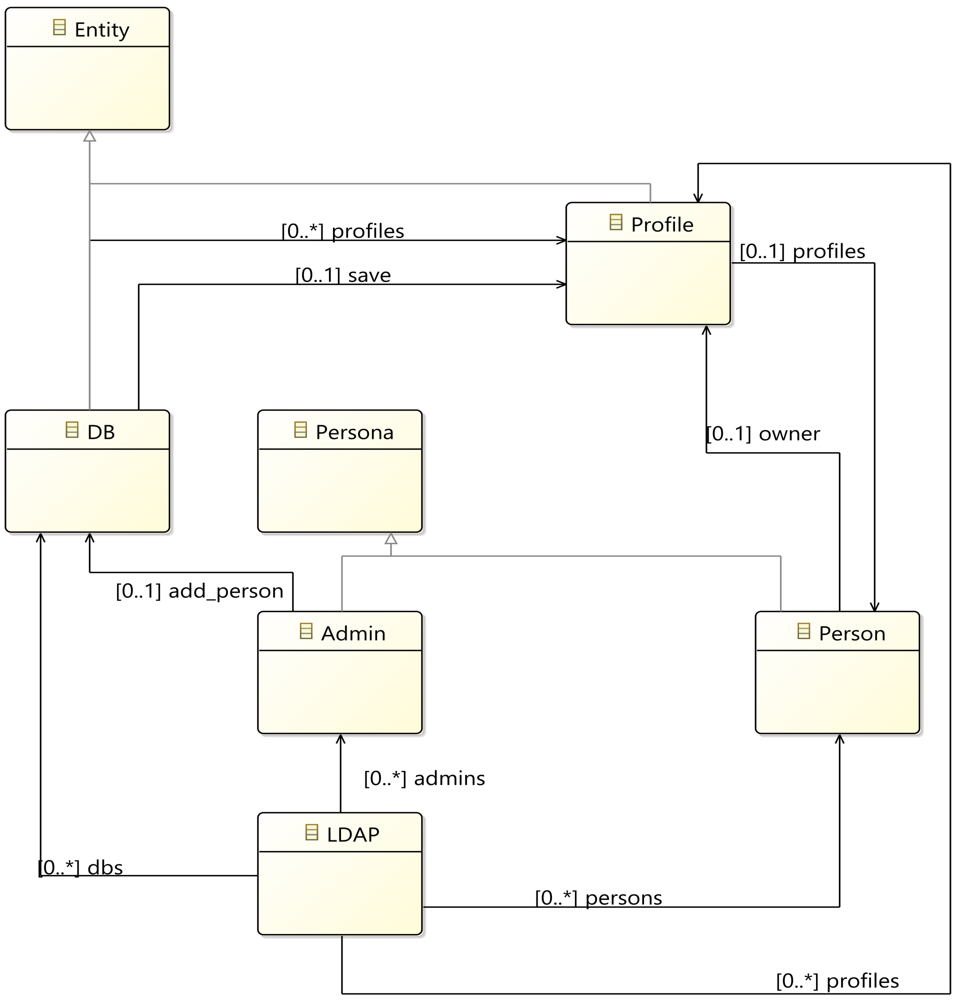


Figure 20: Henshin Class Model LDAP

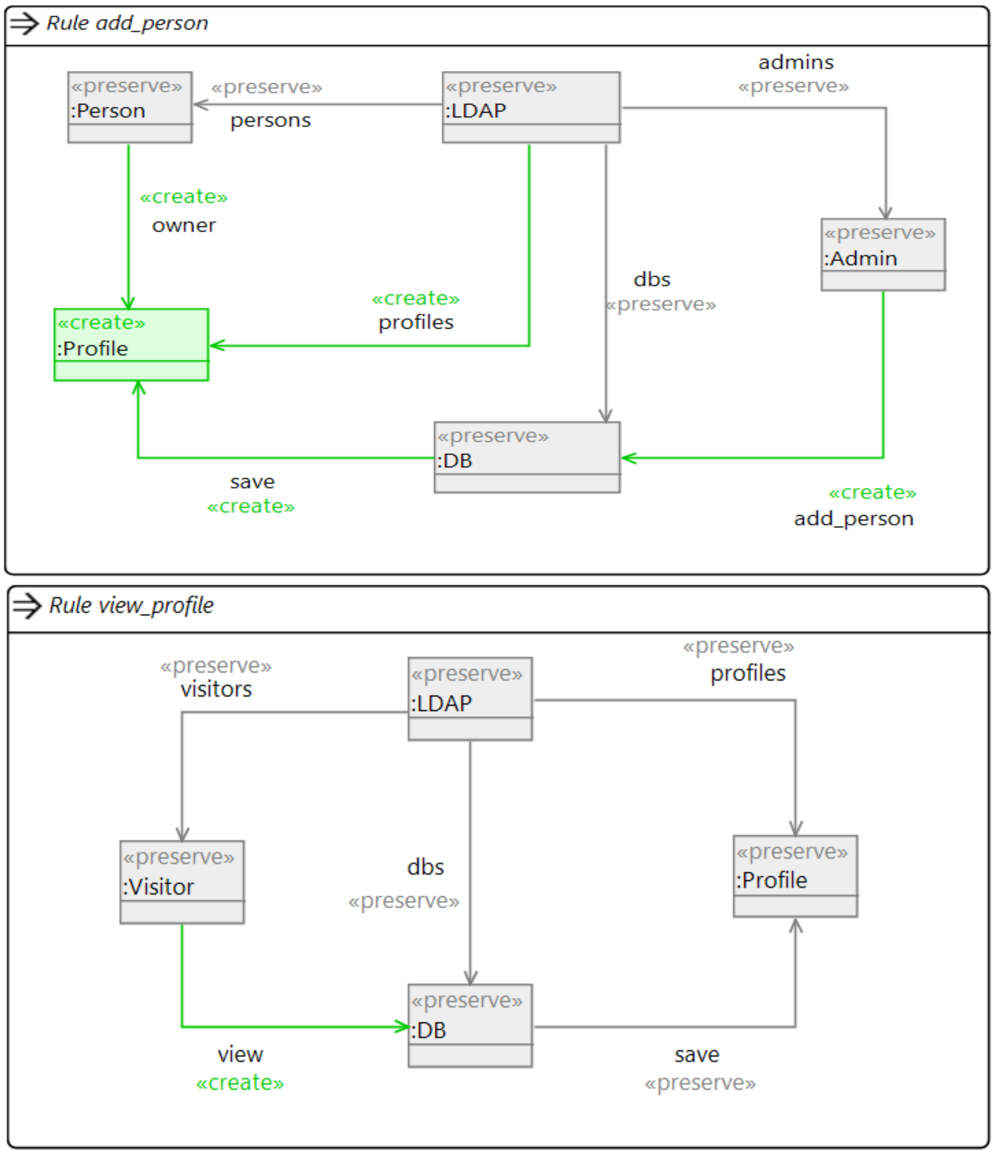
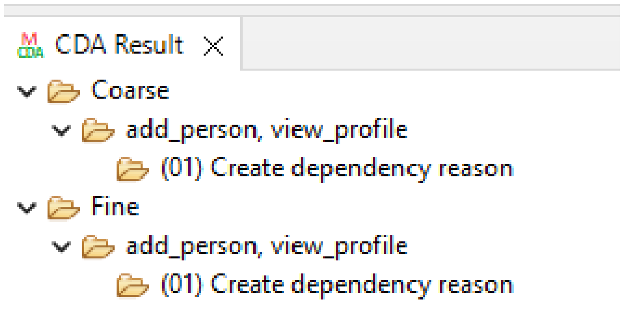
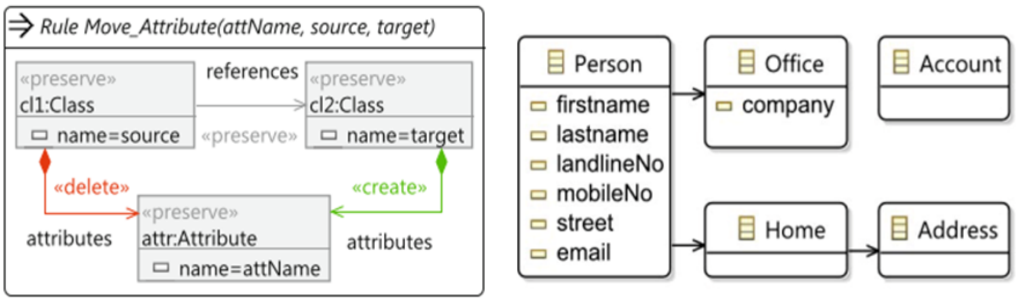


Figure 21: Illustrated rules in Henshin: view_profile rule related to US_2 and add_person related to US_1



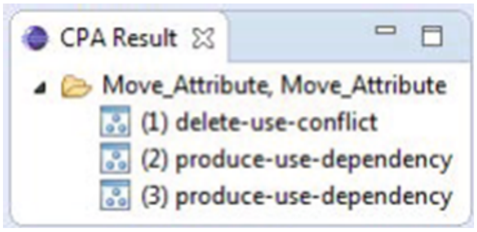
CDA Result
- Coarse
  - add_person, view_profile
    - (01) Create dependency reason
- Fine
  - add_person, view_profile
    - (01) Create dependency reason

Figure 22: Henshin CPA result



(a) (b)

Figure 23: Henshin refactoring rule (a) and class model Address Book (b) [36]


CPA Result

Move_Attribute, Move_Attribute
- (1) delete-use-conflict
- (2) produce-use-dependency
- (3) produce-use-dependency

Figure 24: The result view[8]

subsequent dependency results differ in their target of the second attribute movement. The first produce/use-dependency (2) represents the case of moving the attribute back to the original class, which leads to a smaller minimal model with only two classes referencing each other, as depicted in Figure 25. The highlighting by enclosing hash marks is the most important information, since the enclosing element is the cause of the dependency. The link between 2:Class and 3:Attribute is created by the first rule application and is required by the second application. Since all elements and values in the minimal model may be matched by the first and the second rule application, there is a generic approach to represent attribute values. Value *r1_source_r2_target*, e.g., means that it must conform to value source in rule *r1* and value target in rule *r2*, respectively (compare Fig. 23(a)).

The second dependency reported in Figure 24 is the handling of two consecutive attribute shifts.

5.3 Comparative Analysis

In the following we compare two graph-based modelling tools using different categories, inspired by [16].

- Area: Which Area does the tools support?
- Typing: What is the typing mode that the tools operate in?
- Control: How do the tools provide control over the application of rules?
- Strategy: What strategies do the tools employ for analysis and conflict resolution?
- Relevant Features: What advanced features and capabilities do the tools offer?
- Interface: How user-friendly are the tools' interfaces?

An overview of this comparison is given in table 8. Finally, we choose one of these tools for our approach namely analysis of conflicts and dependencies between user stories.

As a result of the introduction of the robust CPA (critical pair analysis) extension within the Henshin framework, our choice has shifted towards adopting Henshin over GROOVE. This strategic decision is underpinned by the unique capabilities of the CPA

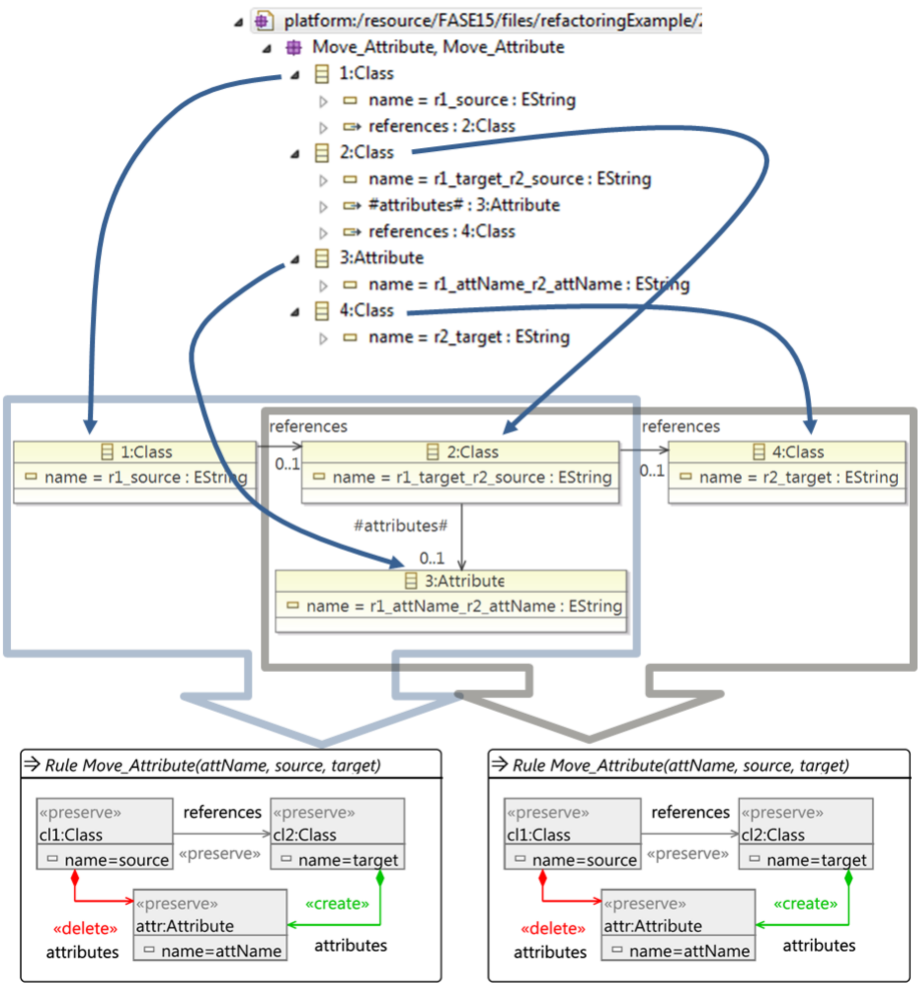

platform:/resource/FASE15/files/refactoringExample/
Move_Attribute, Move_Attribute
1:Class
name = r1_source : EString
references : 2:Class
2:Class
name = r1_target_r2_source : EString
#attributes# : 3:Attribute
references : 4:Class
3:Attribute
name = r1_attName_r2_attName : EString
4:Class
name = r2_target : EString
references
0..1
#attributes#
Rule Move_Attribute(attName, source, target)
«preserve»
cl1:Class
name=source
cl2:Class
name=target
attr:Attribute
name=attName
«delete»
«create»
attributes

Figure 25: Detailed representation of a critical pair showing a dependency [8]

Aspects	GROOVE	Henshin
Area	Model transformation, Verification, Analysis, Simulation	EMF-based transformation model, Analysis, Simulation, Verification
Typing	Node types, Full types	Full types
Control	Rule priorities, Control program	non-deterministic rule choices, sequential and priorities over Transformation unit
Strategy	Linear exploration, Full state space exploration, Find rule application, Random linear exploration	model analysis and conflict resolution
Relevant Features	Quantified rules, Wildcards and regular expressions	CPA extension of Henshin
Interface	GUI helps debugging, GUI helps prototyping, GUI helps analysis of results	GUI helps prototyping, GUI helps analysis of results
Strong Points	Rapid prototyping, Local confluence check, Analysis capabilities, Rule expressiveness	Rapid prototyping, supporting analyzing of conflicts and dependencies, Interoperability using API, Scalability, Performance
Weak Points	Model trafo support, Interoperability, Scalability, Performance, Animation, Analyzing conflicts and dependencies	required learning curve for new users to EMF and transformation process, complexity in rule sequencing, not support untyped mode

Table 8: Overview of graph-based modeling tools inspired by [16]

extension, which empower us to comprehensively evaluate conflicts and dependencies within graph-based transformation rules associated with USs.

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.

5.4 Conclusion

It is imperative to note that Groove conducts its analysis with regard to particular instance graphs, whereas Henshin's (critical pair analysis (CPA) remains agnostic to the specifics of instance graphs. Consequently, model checking in Groove necessitates the entire state space, whereas 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. Usually, the state space in such systems becomes infinitely large, which is a challenge for GROOVE.

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

In the Groove approach, it is not clearly defined what dependencies and conflicts mean. In the case of dependencies, the order in which rules are applied is specified and this order is expected to be maintained in the state space. If there is no alternative order, it can be concluded that there is a dependency between the rules. Conflicts are similar: if rules can be applied in a graph and the resulting transitions are compatible with each other, there is no conflict. However, if the transitions are incompatible, there is a conflict. However, the analysis of the state space always depends on the instance graph selected at the beginning.

6 Conclusion

In Section 2, we have undertaken a comparative analysis of various techniques for evaluating the quality of user stories, categorized according to different criteria. The INVEST criteria are deemed applicable in manual environments, such as those overseen by Product Owners, and necessitate a manual assessment against these criteria. The QUS framework, on the other hand, has been implemented through a tool named AQUSA to automate the process of assessing the quality of user stories. In our workflow's initial phase, we employ the QUS framework and AQUSA as tools to scrutinize user stories within the backlog, ensuring their applicability and alignment for subsequent actions.

In Section 3, we conducted an experiment to assess the performance of the Visual Narrator, GPT-3.5, and a CRF-based approach in automating the extraction of domain concepts from agile product backlogs. Given that the CRF generates a graph-based model, it is particularly advantageous for our approach, serving as input for the development of a transformation rule system.

Section 4 revolves around the comparison of various lexical resource techniques for computation. VerbNet, specializing in verbs, FrameNet, with a broader spectrum encompassing nouns and adjectives, and WordNet, offering a wide array of words spanning various parts of speech, have been evaluated.

For our purposes, VerbNet stands out as the most suitable technique. Its hierarchical classification of verbs into classes provides a structured and comprehensive approach for categorizing a wide range of verbs based on their semantics. This is of utmost importance in our endeavor to formulate transformation rules rooted in semantic interpretations of actions within user stories.

Finally, in Section 5, we scrutinize graph transformation tools, specifically Henshin and GROOVE. GROOVE is best suited for comprehensive analysis of the full state space and random-linear exploration, proving highly effective for analytical and verification purposes. Henshin, on the other hand, is an EMF-based transformation model boasting scalability and interoperability as its key attributes.

Notably, Henshin supports the analysis of conflicts and dependencies using the CPA extension. Furthermore, a persuasive attribute in favor of Henshin lies in its intrinsic provision of a versatile Application Programming Interface (API) through the CPA extension.

Owing to the utilization of various tools and techniques for ascertaining dependencies and conflicts among user stories, several pertinent questions arise: Can we succeed in classifying all verbs in roughly into three categories namely "create", "delete" and "forbid"? With regard to the quality analysis presented, there are not suitable analyses for all criteria, what could the analysis for missing criteria look like? Can we use graph transformations to find conflicts and dependencies between annotated user stories? To what extent does the effectiveness of our approach hinge on the datasets provided within the backlogs? would we characterize the overall effectiveness of our approach?

References

[1] A. Al-Hroob, A. T. Imam, and R. Al-Heisa. The use of artificial neural networks for extracting actions and actors from requirements document. *Information and Software Technology*, 101:1–15, 2018.

[2] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I 13*, pages 121–135. Springer, 2010.

[3] T. Arendt and G. Taentzer. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering*, 20:141–184, 2013.

[4] M. Arias, A. Buccella, and A. Cechich. A Framework for Managing Requirements of Software Product Lines. *Electronic Notes in Theoretical Computer Science*, 339:5–20, July 2018. ZSCC: 0000025.

[5] S. Arulmohan, M.-J. Meurs, and S. Mosser. Extracting Domain Models from Textual Requirements in the Era of Large Language Models. MDEIntelligence (co-located with ACM/IEEE 26th International Conference on . . . , 2023.

[6] C. F. Baker, C. J. Fillmore, and J. B. Lowe. The berkeley framenet project. In *COLING 1998 Volume 1: The 17th International Conference on Computational Linguistics*, 1998.

[7] D. M. Berry and E. Kamsties. Ambiguity in requirements specification. In *Perspectives on software requirements*, pages 7–44. Springer.

[8] K. Born, T. Arendt, F. Heß, and G. Taentzer. Analyzing conflicts and dependencies of rule-based transformations in Henshin. In *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 18*, pages 165–168. Springer, 2015.

[9] K. Born and G. Taentzer. An algorithm for the critical pair analysis of amalgamated graph transformations. In *International Conference on Graph Transformation*, pages 118–134. Springer, 2016.

[10] L. Buglione and A. Abran. Improving the user story agile technique using the invest criteria. In *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*, pages 49–53. IEEE, 2013.

[11] E. Cambria and B. White. Jumping NLP curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[12] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, and J. N. och Dag. An industrial survey of requirements interdependencies in software product release planning. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 84–91. IEEE, 2001.

[13] M. Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[14] F. Dalpiaz, A. Ferrari, X. Franch, and C. Palomares. Natural Language Processing for Requirements Engineering: The Best Is Yet to Come. 35:115–119, 2018.

[15] J. Doe. Recommended practice for software requirements specifications (ieee). *IEEE, New York*, 2011.

[16] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14:15–40, 2012.

[17] M. Glinz. Improving the quality of requirements with scenarios. In *Proceedings of the second world congress on software quality*, volume 9, pages 55–60, 2000.

[18] A. Gomez, G. Rueda, and P. P. Alarcón. A systematic and lightweight method to identify dependencies between user stories. In *Agile Processes in Software Engineering and Extreme Programming: 11th International Conference, XP 2010, Trondheim, Norway, June 1-4, 2010. Proceedings 11*, pages 190–195. Springer, 2010.

[19] D. Greer and G. Ruhe. Software release planning: an evolutionary and iterative approach. 46:243–253, 2004.

[20] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[21] A. C. D. H. J. Hartmanis, T. Henzinger, J. H. N. J. T. Leighton, and M. Nivat. Monographs in Theoretical Computer Science An EATCS Series. 2006.

[22] J. H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proceedings of the 24th international conference on software engineering*, pages 105–115, 2002.

[23] P. Heck, M. Klabbers, and M. van Eekelen. A software product certification model. *Software Quality Journal*, 18:37–55, 2010.

[24] P. Heck and A. Zaidman. A quality framework for agile requirements: a practitioner's perspective. *arXiv preprint arXiv:1406.4692*, 2014.

[25] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. In *Model Checking Software: 13th International SPIN Workshop, Vienna, Austria, March 30-April 1, 2006. Proceedings 13*, pages 299–305. Springer, 2006.

[26] K. Kipper, A. Korhonen, N. Ryant, and M. Palmer. Extending VerbNet with Novel Verb Classes. In *LREC*, pages 1027–1032, 2006.

[27] A. Korhonen and T. Briscoe. Extended lexical-semantic classification of English verbs. In *Proceedings of the Computational Lexical Semantics Workshop at HLT-NAACL 2004*, pages 38–45, 2004.

[28] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In C. E. Brodley and A. P. Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 282–289. Morgan Kaufmann, 2001.

[29] B. Levin. *English verb classes and alternations: A preliminary investigation*. University of Chicago press, 1993.

[30] O. I. Lindland, G. Sindre, and A. Solvberg. Understanding quality in conceptual modeling. *IEEE software*, 11(2):42–49, 1994.

[31] O. Liskin, R. Pham, S. Kiesling, and K. Schneider. Why we need a granularity concept for user stories. In *Agile Processes in Software Engineering and Extreme Programming: 15th International Conference, XP 2014, Rome, Italy, May 26-30, 2014. Proceedings 15*, pages 110–125. Springer, 2014.

[32] K. Logue and K. McDaid. Handling uncertainty in agile requirement prioritization and scheduling using statistical simulation. In *Agile 2008 Conference*, pages 73–82. IEEE, 2008.

[33] P. Lombriser, F. Dalpiaz, G. Lucassen, and S. Brinkkemper. Gamified requirements engineering: model and experimentation. In *Requirements Engineering: Foundation for Software Quality: 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings 22*, pages 171–187. Springer, 2016.

[34] G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, and S. Brinkkemper. Improving agile requirements: the quality user story framework and tool. *Requirements engineering*, 21:383–403, 2016.

[35] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper. Forging high-quality User Stories: Towards a discipline for Agile Requirements, 2015.

[36] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling*, 6:269–285, 2007.

[37] G. A. Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[38] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller. Introduction to wordnet: An on-line lexical database. *International journal of lexicography*, 3(4):235–244, 1990.

[39] M. Moens, M. Steedman, et al. Temporal ontology and temporal reference., 2005.

[40] S. Mosser, C. Pulgar, and V. Reinharz. Modelling Agile Backlogs as Composable Artifacts to support Developers and Product Owners. *Journal of Object Technology (JOT)*, 2022.

[41] U. of Colorado. VerbNet Guidelines, 2012.05.09.

[42] E. Paja, F. Dalpiaz, and P. Giorgini. Managing security requirements conflicts in socio-technical systems. In *Conceptual Modeling: 32th International Conference, ER 2013, Hong-Kong, China, November 11-13, 2013. Proceedings 32*, pages 270–283. Springer, 2013.

[43] C. A. Prolo. Generating the XTAG English grammar using metarules. In *COLING 2002: The 19th International Conference on Computational Linguistics*, 2002.

[44] I. K. Raharjana, D. Siahaan, and C. Fatichah. User stories and natural language processing: A systematic literature review. *IEEE access*, 9:53811–53826, 2021.

[45] A. Rensink. Representing first-order logic using graphs. In *International Conference on Graph Transformation*, pages 319–335. Springer, 2004.

[46] A. Rensink and J.-H. Kuperus. Repotting the geraniums: on nested graph transformation rules. *Electronic Communications of the EASST*, 18, 2009.

[47] M. Robeer, G. Lucassen, J. M. E. M. van der Werf, F. Dalpiaz, and S. Brinkkemper. Automated Extraction of Conceptual Models from User Stories via NLP, 2016.

[48] W. N. Robinson. Integrating multiple specifications using domain goals. *ACM SIGSOFT Software Engineering Notes*, 14(3):219–226, 1989.

[49] T. Sedano, P. Ralph, and C. Péraire. The product backlog. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 200–211. IEEE, 2019.

[50] M. Tichy, C. Krause, and G. Liebel. Detecting performance bad smells for henshin model transformations. *Amt@ models*, 1077, 2013.

[51] G. Varró, F. Deckwerth, M. Wieber, and A. Schürr. An algorithm for generating model-sensitive search plans for EMF models. In *Theory and Practice of Model Transformations: 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings 5*, pages 224–239. Springer, 2012.

[52] Y. Wautelet, S. Heng, S. Kiv, and M. Kolp. User-story driven development of multi-agent systems: A process fragment for agile methods. 2017.

[53] Y. Wautelet, S. Heng, M. Kolp, and I. Mirbel. Unifying and extending user story models. In *Advanced Information Systems Engineering: 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings 26*, pages 211–225. Springer, 2014.

[54] Y. Wautelet, S. Heng, M. Kolp, and I. Mirbel. Unifying and Extending User Story Models, 2014.

[55] M. Wynne, A. Hellesoy, and S. Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[56] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30, 1997.