Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
AG Softwaretechnik

Analysis of Conflicts and Dependencies between User Stories using Graph Transformation

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science

vorgelegt von
Amir Rabieyan Nejad
Matrikel-Nr: 3350269

26. März 2024

1 Analysing Redundancy

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

In Section 1.1 we illustrate the requirements and functional needs that are used as input to the design phase to satisfy and bring value and benefit to the stakeholder. In the Section 1.2 we explain the design decisions of the workflow shown in **??** and we explain how the architecture is structured and what the components and their classes look like. In Section 1.3 comes the implementation to show what we have implemented and in Section **??** we show how we have tested it. Finally, we will apply this approach to 19 backlogs entered with the CRF tool[1] we enter all backlogs as input and the results are evaluated in section**??**.

1.1 Requirements

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

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

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

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

1.2 Design

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

To fulfil the requirements mentioned in 1.1, we use the CRF tool [**?**] for the annotation of backlogs and the Henshin API [**?**] under Java programming language for the automatic generation of rules for each US in the backlog. Moreover, we use Henshin's CDA(conflict and dependency analysis) feature [**?**] to automatically recognise conflicting US-pairs.

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

CRF Tool

As input, we receive a graph-based model generated by the CRF tool, which represents the refined and annotated dataset for the recognition of *entities*, *actions*, *persons* and *benefits* of USs [**?**]. Mooser et al. have linked each *Persona* to each primary *Action* (as *Trigger* relationships), each primary *Actions* to each primary *Entity* (as *Target* relationships) and each primary/secondary *Entity* to each primary/secondary *Entity* (implying a *Contains* relationship)[**?**]. As output we receive a JSON-file which contains all annotated USs separated by each backlogs.

Identifying USs in JSON-File

Annotated USs in each JSON-file have no identifier. To distinguish user stories, we use a Python script called *nummerize_us.py* [2], which receives JSON-files as input and adds a JSON-object called "US_Nr" with a number as value to each US and returns the JSON-files as output.

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

Creating Ecore Meta-Model

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

Accordingly, we create an Ecore meta-model as shown in Figure 1, which is inspired by the meta-model shown in Figure **??** and corresponds to the JSON-objects in the JSON-file as follows:

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

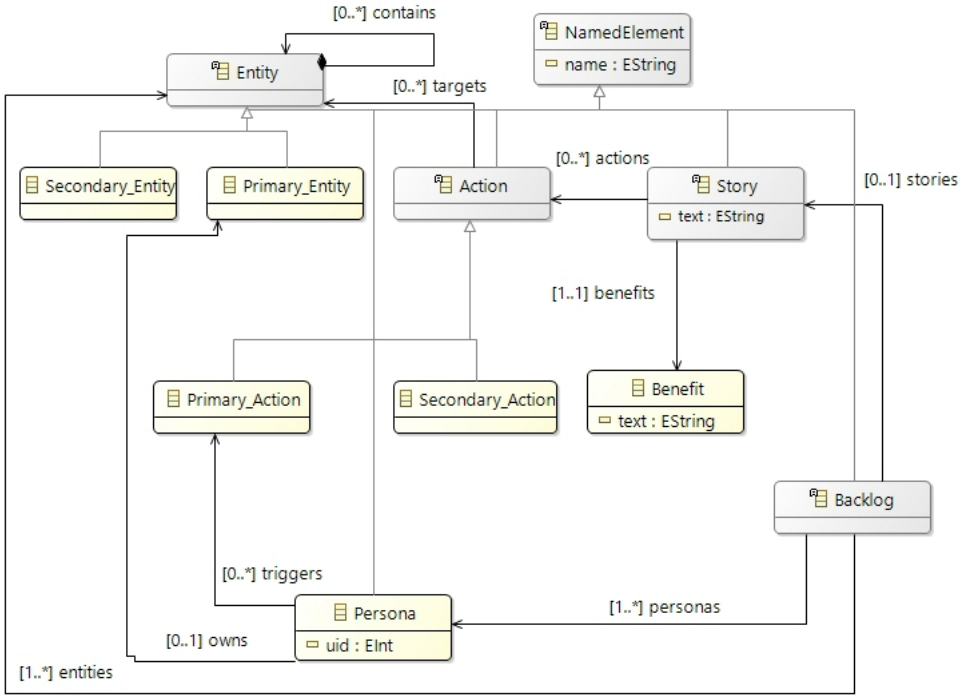

Figure 1: Ecore meta-model inspired by Mosser et al. [**?**]

Creating Rules

With the identified USs in the the JSON-file, we generate rules with the Henshin package *org.eclipse.emf.henshin.model. compact*, which is responsible for the creation of *Transformation rules* and their *Classes*, *Attributes*, *Edges* and annotates them with <*Delete*>, <*Create*>or <*Preserve*>, which are crucial for the CDA tool to recognise the redundant pairs. To generating rules we create a package named *package org.henshin.backlog.code.rule* and specially the class *RuleCreator* which used following classes[3]:

- *org.eclipse.emf.henshin.model.compact.CModule*: CModule class can import elements from an Ecore file to use them in the transformation process responsible for linking the Ecore meta model to the Henshin-file to be created.
- *org.eclipse.emf.henshin.model.compact.CRule*: Once we have a CModule, we can specify transformation rules with the CRule class and create them.
- *org.eclipse.emf.henshin.model.compact.CNode*: Now that we have a transformation rule, we want to fill this rule with nodes. To create a node within a transformation

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

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

- org.henshin.backlog.code.rule.RuleCreator: We implement RuleCreator class that creates a rule with annotated nodes, edges and attributes based on a JSON-file as input and a Henshin-file containing all rules as output, where each rule and its elements correspond to the individual US and their JSON-objects/arrays in the JSON-file. The most important design decision of this class is the way attributes and edges are annotated to apply CDA to Henshin-files. We decided to annotate the "name" attribute of all Primary/Secondary Actions/Entities and their associated edges including targets, triggers and contains in <delete >action. The main goal is to increase the probability of finding the US-pairs that have the same Action/Entity and target the same Entity, which can be a potentially redundant US-pair.

Methods of the RuleCreator Class

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

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

2. assingCmodule: This method assign a CModule to a Ecore meta-model. It creates a new CModule object with the provided Henshin-file name, adds imports from the Ecore file, and returns the module.

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

4. processRule: It takes the "US_Nr" JSON-object as input and creates a new CRule with the name of unique US identifier in the CModule.

5. processPersona: It receives as input the persona extracted from the JSON data and the associated CRule to create a new CNode representing the persona within the provided CRule and adds the attribute "name" with persona as value. Finally, the created CNode representing the persona is returned.

6. processText: It receives as input US text extracted from JSON data and the associated CRule to create a new CNode representing the text within the provided CRule and adds the attribute "text" with US text as value. Finally, the created CNode representing the US text is returned.

7. processActions: The processActions method is responsible for creating CNode objects that represent actions within the CModule. As parameters, it receives the JSON-object of the actions, the CNode-object representing the persona associated with the actions and the unique identifier of the US. Since the edge triggers only refer to the persona and the primary action, a new CNode is created for each primary action that represents the primary action within the provided CRule, an attribute "name" is added and an edge is created from the persona node to the primary action with the label "triggers". For each secondary action, a new CNode is created to represent the action within the provided CRule and an attribute "name" is added to the action node.

8. checkEntityIsTarget: It receives the name of the entity and the JSON-array with information about target edges. The method iterates through the JSON-array targets, which contains arrays that represent targets edges between actions and entities. It compares the targets entity with the specified entity. If there is a match, it returns true to indicate that the entity is a target.

9. processEntities: It receives as parameters the JSON-object with information about the entities, the CRule object representing the US to which the entities belong and the JSON-array with information about the targets associated with the entities. The method checks whether primary/secondary entities are present, then creates a CNode for each primary/secondary entity and checks whether the entity is present in the target array. If this is the case, its attribute "name" is annotated for deletion. This method is used within the processContainsEdges method to determine whether an entity involved in a contains relationship is also a target of another entity.

10. processContainsEdges: It receives the JSON-object to be processed, the JSON array with information about contains/target edges and the US identifier as parameters. It first checks whether both entities belong to contains edges. If both entities exist, an edge is created between them in CRule with the name "contains". If one of the entities is a target of another entity (as specified in the targets array), the edge is annotated for deletion. If none of the entities is a target, the edge is annotated as "preserve".

Conflicting and Dependency Analysis

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

Since the CPA API [4] does not yet take into account conflicts and dependencies related to *attribute*, we decided to use the user interface (UI) of the CPA extension of Henshin , which supports domain experts in the development of rules through the interactive use of CPA.

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

To apply CDA to Henshin files, we just need to right-click on the Henshin file and select *Henshin>conflict and Dependency Analysis* from the context menu as shown in Figure 2. A user interface then appears, prompting to select the rule sets to be analysed

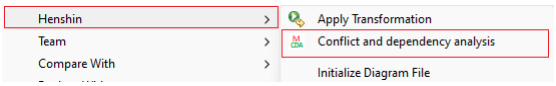

Figure 2: Applying CDA to the selected Henshin file

and the type of analysis. We then select as *First* and *Second Rules* all US rules and as the type of analysis we select *conflicts* as illustrated in Figure 3 On the next page

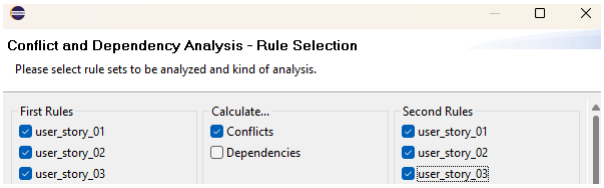

Figure 3: CDA user interface: Selection of rules and type of analysis

of the CDA user interface shown in Figure 4, we specify the depth of analysis that we use with *Fine* granularity when selecting *Create a complete result table* and *Create an abstract result table*. We choose Fine granularity as the depth of analysis due to the fact that it shows all conflict reasons for each conflicting rule pair. A conflict reason is a model fragment whose presence leads to a conflict. General conflict reasons result from different combinations of minimal conflict reasons[**?**].

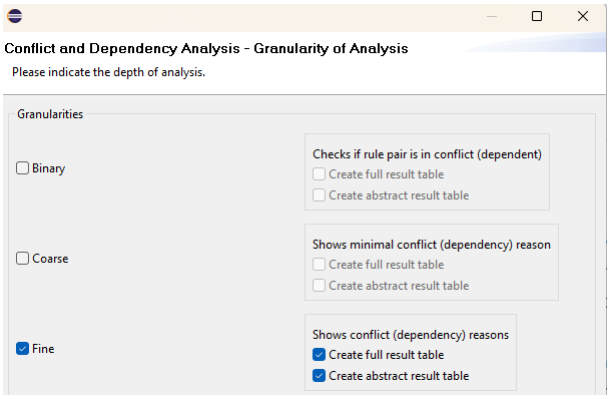

Figure 4: CDA user interface: Selection of report granularity

During the execution of the CDA analysis, the rule pair is analysed and a conflict analysis is performed. Once the calculation is complete, the results are listed in the CDA ->Result window, as shown in Figure 5. The top entry shows the granularity, which in our case is Fine. These entries contain the rule pairs that conflict with each other. Each rule pair contains a number of conflicts.

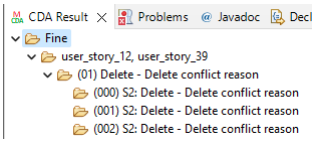

Figure 5: CDA report with fine granularity

Figure 6 shows how the data is saved in the project tree view. The results folder is created in the folder containing the Henshin that was used for the analyses. The new folder name is the date and time at which the analysis was performed. In contrast to the *CDA/Results* view, this folder contains all reasons and atoms together in a rule pair folder.

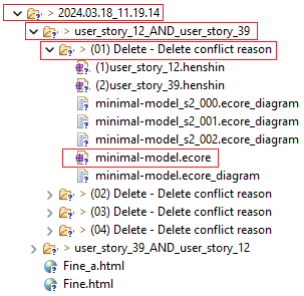

Figure 6: Saving CDA results data in the project structure view

For each conflict reason, there is a *minimal-model.ecore* that contains packages in which various conflict elements such as attributes and references (edges) are mapped together and displayed in different packages. Figure 7 shows the representation of the conflicting attributes and references. An attribute has the property of changing the value and is represented by an arrow ->. The attribute from the first rule is separated from the second rule by an underscore, just as with the nodes.

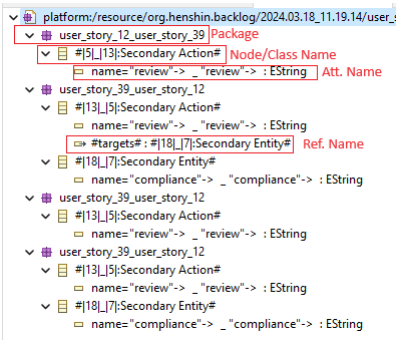

Figure 7: Representation of conflicting attributes and references in *minimal-model.ecore* file

Extracting Textual Report

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

- A table of potential redundant pairs with the number of total redundancy clauses.
- Founded potential redundant US-pairs
- Redundancy words and clauses of founded US-pairs
- Text of US-pairs whose redundancy words are marked with a hash symbol (#).
- Parts of the sentence in which words and clauses are found.

Listing 1 illustrates the format of the textual report. In this case, the report only contains one US-pair.

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

       us_12  us_39
us_12  0      1
us_39  1      0

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

Redundant clauses within user stories are:
* Secondary Action: review
* Secondary Entity: compliance
* Targets: Link from "review" to "compliance" is found.

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

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

The following sentence parts are candidates for possible
 redundancies between user stories:

user_story_12:  so that i can #review# the for #compliance#
 and subsequently approved or denied.
user_story_39:  so that i can #review# them for #compliance#
 and either approve, or fail or deny the plans and record any
```

```
conditions, clearances, or corrections needed from the
applicant.
```

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

In order to extracting a textual report associated with a specific backlog, we implement a class called *ReportExtractor* within the package *org.henshin.backlog.code.report*, which include the following classes form the package *org.eclipse.emf.ecore*[5]. These classes are important for reading the content of minimal-model.ecore:

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

Storing Redundancy Items into RedundancyItems Class

The following classes were created to represent the extracted model object accordingly. All these classes are extensions of the class *RedundancyItems*, which contains all extracted model object from minimal-model.ecore such as EClass, EAttribute or EReference:

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

- PrimaryAction/SecondaryAction: Which has only saved the EClass specified by "Primary/Secondary Action" and the E-Attribute model object with the methods *getType* to retrieve the saved EClass and *getName* to retrieve the saved E-Attribute.
- PrimaryEntity/SecondaryEntity: Which has only saved the EClass specified by "Primary/Secondary Entity" and the E-Attribute model object with the methods *getType* to retrieve the saved EClass and *getName* to retrieve the saved E-Attribute.
- Targets: The EClass specified by "Primary/Secondary Action" as *outgoing edge* and an E-Attribute model object as *incoming edge* with "Primary/Secondary Entity". The method *getType* retrieve the stored EClass and method *getName* retrieve the stored E-Attribute.
- Contains: The EClass specified by "Primary/Secondary Entity" as *outgoing edge* and an E-Attribute model object as *incoming edge* with "Primary/Secondary Entity". The method *getType* retrieve the stored EClass and method *getName* retrieve the stored E-Attribute.
- Triggers:
- Contains: The EClass specified by "Persona" as *outgoing edge* and an E-Attribute model object as *incoming edge* with "Primary Action". The method *getType* retrieve the stored EClass and method *getName* retrieve the stored E-Attribute.
- RedundantPair: Stores the identifier of the two created user stories as a redundant pair. It also saves the total number of redundancy clauses within the US-pair.
- TargetsPair: Stores effective value of *Primary/Secondary Action* and *Primary/Secondary Entity* in action and entity fields accordingly.
- ContainsPair: Stores effective value of *Primary/Secondary Entity* as *parent/child entity* due to the fact that parent entity is a containment of child entity.
- TriggersPair: Stores effective value of *Persona* as a persona and *Primary Action* as an action.

The calss RedundancyItems contains following methods which are crucial for class ReportExtractor:

- isInCommonContains: This method is designed to determine whether a given entity (specified by its name) is part of a redundant pair listed in the Contains array of related user stories stored in a JSON file. As input, it receives the name of the entity for which we want to check whether it exists in a redundant pair. In addition, a list of redundant pair objects that represent pairs of entities where one entity contains the other. If there is a match with the parent entity, it returns the child entity and vice versa.

- isInCommonTargets: This method is responsible for determining whether a particular action/entity is part of a redundant pair listed in the "Targets" array of related user stories stored in a JSON file. As input, it receives the name and type of the first element, which is an action, and the name and type of the second element, which is an entity. It also receives a list of target pair objects, which are pairs of common actions and entities between US-pairs. For each pair, it checks whether the specified names and types match either the action or entity in the pair. If a match is found, the method returns *true*, which means that the specified elements are part of a redundant pair.

- printRedundantItems: This method is responsible for generating a report of redundancy items based on the data stored in the class instance. It takes several parameters, including a FileWriter for writing to a file, lists of various pairs of targets, contains, and triggers, and a JSONObject for storing the report data in JSON format which is crucial for evaluation.

ReportExtractor Class: Methods related to Extracting Report

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

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

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

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

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

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

   In addition, the methods *hasEntitys*, *hasActions* and *hasTargets* are used to check whether the identified elements contain primary/secondary actions and primary/secondary entities and targets. If the identified elements fulfil the criteria, the redundancy pair is included in the report.

It then writes the potentially redundant USs and their clauses to the text report file. It also extracts and saves relevant information in JSON format for further analysis.

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

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

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

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

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

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

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

6. iteratePackages:This method identifies redundancies within the attributes and references of EClasses in a minimal model, saves them accordingly and updates the RedundancyItems object.

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

   It iterates through every EClassifier in the minimal package that contains EClasses and checks whether EClass "#" is present; if this is the case, the conflict was detected by CDA. If an attribute is found, the class of the conflicting attribute is determined and added to the corresponding section within RedundancyItems (e.g. Primary/Secondary Action/Entity).

   Each EReference is then iterate through in the EClass. Depending on the reference name, the reference is added to the corresponding section within RedundancyItems

(e.g. Triggers, Targets, Contains). The method is completed once all EClassifiers within the specified EPackage have been processed.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    It iterates through the list of common target pairs and checks whether both elements of the target pair are present in both segments. It then increments the

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

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

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

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

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

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

    It checks whether both USs contain a main clause part or whether one of them has a benefit part or whether both USs also have a benefit part. It applies hash symbols to common elements that only occur in the part of the sentence that occurs in the same segment (e.g. only main or only benefit part of the sentence).

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

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

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

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

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

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

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

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

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

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

ReportExtractor Class: Methods related to Creating Table

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

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

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

   It reads the existing content of the report file into a StringBuilder. It creates a table to display the potential redundancies between USs and the count of total redundancy elements. The table headers and contents are generated based on the redundant pairs. It calculates the maximum width for each column in the table to ensure proper formatting. Finally, the table content is written to the FileWriter, followed by the existing content stored in the report's StringBuilder.

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

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

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

3. createTable: This method prepares the content for the table in which potential redundancies between user stories are displayed, taking into account the total redundancy count between each pair of user stories based on the RedundantPair objects provided.

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

   It initialises a two-dimensional array containing the contents of the table. The size of the table is determined by the count of unique pairs of USs plus one for the header row and column. It fills the header row and the first column of the table with unique pairs of US-pairs, replacing "user_story" with "us" for the purpose of brevity. It calculates the maximum redundancy count between each pair of USs by calling the method *getTotalRedundanciesFromPair*. Finally, it fills the table with the total redundancy count.

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

1.3 Implementation

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

The entire implementation is available in the GitHub repository [6].

Objective and Scope

The goal and scope of the work is divided into three phases. Firstly, converting the USs annotated by the CRF tool into graph transformation rules; secondly, using the CDA function of the Henshin to automatically report redundancy between USs; thirdly, extracting important information from the CDA report into a text report. For further analysis, we stored the information in a JSON file to be able to import the data into another platform such as MS Excel. Figure 8 illustrates the implementation phases mentioned.

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

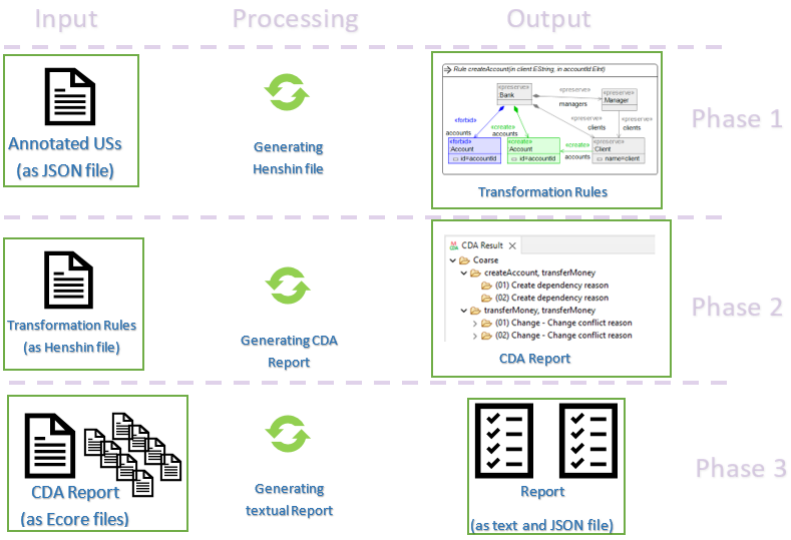

Figure 8: Three implementation phases

Methodology

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

- Eclipse as IDE: Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base work workspace and an extensible plug-in system for customizing the environment.
- Eclipse Modeling Project[7]: The Eclipse Modeling Project focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modelling frameworks, tooling, and standards implementations.
- Eclipse Modeling Framework (EMF)[8]: The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.
- Henshin[9]: Henshin is an in-place model transformation language for the Eclipse Modeling Framework (EMF). It supports direct transformations of EMF model instances (endogenous transformations), as well as generating instances of a target language from given instances of a source language (exogenous transformations)

[7]https://eclipse.dev/modeling/
[8]https://eclipse.dev/modeling/emf/
[9]https://wiki.eclipse.org/Henshin

- Henshin's CDA feature[10]: Henshin's conflict and dependency analysis feature enables the detection of potential conflicts and dependencies of a set of rules.
- GitHub as version control: GitHub is a developer platform that allows developers to create, store, manage and share their code. It uses Git software, providing the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project.

Data Structures

Error Handling

User Story Structure

Limitations

Following limitation which at the beginning should be clarify:

- We have to use Eclipse version 2023-03 because Henshin files cannot be installed with the latest version of Eclipse.
- We have to work with Java because all Henshin files APIs are not available in other programming languages like Python.
- CDA API is not yet implemented for considering relationships and dependencies between attributes. This forces us to use the graphical CDA interface instead of the CDA API.
- Lack of Henshin documentation regarding methods and classes, which makes it time consuming to understand the methods and make the right decision.

1.4 Test

In this section, we aim to validate certain functionalities, check the system requirements and ensure reliability and robustness of implemented classes and methods. As a test strategy, we perform unit tests with *JUnit* version 4[11] as version 4 is more suitable and compatible with Eclipse version 2023-03.

[10] https://wiki.eclipse.org/Henshin/Conflict_and_Dependency_Analysis
[11] https://junit.org/junit4/

Test Environment Configuration

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

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

Scope of Testing

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

Test Cases

This section describes the specific test cases that are performed during the tests. Each test case contains a description of the test scenario, the data provided and the expected result. Table 1 illustrates the test cases for the class RuleCreator_Test.java and table 2 illustrates the test case for the class ReportExtractor.java.

Test Case	Supplied Data	Expected Outcome	Description
testAssignCmodule	Assing a dummy ECore model	Through an exception: *EcoreFileNotFound.class*	Check whether the ECore model already exists and CModule is correctly assigned
UndefindedEntity (testProcessContainsEdges)	Specify an entity that is not contained as "Entity" in the JSON file, but appears as "Contains"	Through an exception: *EntityInJsonFileNotFound.class*	Check whether the entity that appears in the *Contains* entry has already been identified as an entity
ActionNotExist (testProcessJsonFile)	Specify an action that is not contained as *Action* in the JSON file, but appears as *Contains, Targets or Triggers*	Through an exception: *ActionInJsonFileNotFound.class*	Check whether the action that appears in the "Targets", "Contains" or "Triggers" entries has already been identified as an action
ContainsNotExist (testProcessJsonFile)	JSON file with a US without "Contains" entry	Through an exception: *ContainsInJsonFileNotFound.class*	Check whether there is an entry "Contains" in the related US in JSON file
EntityNotExist (testProcessJsonFile)	JSON file with a US without "Entity" entry	Through an exception: *EntityInJsonFileNotFound.class*	Check whether there is an entry "Entity" in the related US in JSON file
PersonaNotExist (testProcessJsonFile)	JSON file with a US without "Persona" entry	Through an exception: *PersonaInJsonFileNotFound.class*	Check whether there is an entry "Persona" in the related US in JSON file
TargetsNotExist (testProcessJsonFile)	JSON file with a US without "Targets" entry	Through an exception: *TargetsInJsonFileNotFound.class*	Check whether there is an entry "Targets" in the related US in JSON file
TextNotExist (testProcessJsonFile)	JSON file with a US without "Text" entry	Through an exception: *TextInJsonFileNotFound.class*	Check whether there is an entry "Text" in the related US in JSON file
TriggersNotExist (testProcessJsonFile)	JSON file with a US without "Triggers" entry	Through an exception: *TriggersInJsonFileNotFound.class*	Check whether there is an entry "Triggers" in the related US in JSON file
UsNrNotExist (testProcessJsonFile)	JSON file with a US without "Us_Nr" entry	Through an exception: *UsNrInJsonFileNotFound.class*	Check whether there is an entry "US_Nr" in the related US in JSON file
PrimaryActionNotFound (testProcessTargetsEdges)	Specify a primary action that is not contained as "Primary Action" in the JSON file, but appears as "Targets"	Through an exception: *ActionInJsonFileNotFound.class*	Check whether the primary action that appears in the *Targets* entry has already been identified as a primary action
PrimaryEntityNotFound (testProcessTargetsEdges)	Specify a primary entity that is not contained as "Primary Entity" in the JSON file, but appears as "Targets"	Through an exception: *EntityInJsonFileNotFound.class*	Check whether the primary entity that appears in the *Targets* entry has already been identified as a primary entity
SecondaryActionNotFound (testProcessTargetsEdges)	Specify a secondary action that is not contained as "Secondary Action" in the JSON file, but appears as "Targets"	Through an exception: *ActionInJsonFileNotFound.class*	Check whether the secondary action that appears in the *Targets* entry has already been identified as a secondary action
SecondaryEntityNotFound (testProcessTargetsEdges)	Specify a secondary entity that is not contained as "Secondary Entity" in the JSON file, but appears as "Targets"	Through an exception: *EntityInJsonFileNotFound.class*	Check whether the secondary entity that appears in the *Targets* entry has already been identified as a secondary entity
UndefindedEntity (testProcessTargetsEdges)	Specify an entity that is not contained as "Entity" in the JSON file, but appears as "Targets"	Through an exception: *EntityInJsonFileNotFound.class*	Check whether the entity that appears in the *Targets* entry has already been identified as an entity
ReadJsonArrayFromFile	Assign a dummy JSON file	Through an exception: *EmptyOrNotExistJsonFile*	Check whether the JSON file already exists.

Table 1: Test cases for RuleCreator class

Test Case	Supplied Data	Expected Outcome	Description
testEmptyDirectroy	Assing a dummy directory	Through an exception: *CdaReport-DirIsEmpty.class*	Check if CDA Report directory is empty
testEmptyJSONFile	Assing an empty JSON dataset file	Through an exception: *EmptyOrNotExistJ-sonFile.class*	Check if JSON dataset file is empty
completeMajorElements_edge (testExtractReports)	Provision of a CDA report for US-pair with exactly one "Targets" edge	Inclusion of this US-pair in the text report with the nodes "Action", "Entity" and "Targets" edge	Verifies *extractReports* method when all major elements are present in the CDA report and the edge case is reached
completeMajorElements_upperEdge (testExtractReports)	Provide a CDA report for a US-pair with at least one "Targets" edge and redundancy elements such as "Triggers"	Generated textual report contains information about Secondary Entities, Secondary Actions, Targets, and "Triggers"	Verifies *extractReports* method when all major elements are present in the CDA report and the upper edge case is reached
notCompleteMajorElements (testExtractReports)	Provide a CDA report for a US-pair without "Targets" edge, but with action and entity	The US-pair should not reported	Verifies the behavior of the *extractReports* methodwhen not all major elements are present in the input data
getBenefitPartRedundanciesElements (testExtractReports)	Provision of a CDA report for a US-pair that only contains redundancy clauses in the "Benefit" part	Check whether the count of redundancy clauses in the benefit part of the USs matches the value "Benefit Part Redundancy Clause" specified in the JSON_Report file	Verifies the behaviour of the *extractReports* method when there are redundancy elements in the benefit part of USs
getMainPartRedundanciesElements (testExtractReports)	Provision of a CDA report for a US-pair that only contains redundancy clauses in the "Main" part	Check whether the count of redundancy clauses in the main part of the USs matches the value "Main Part Redundancy Clause" specified in the JSON_Report file	Verifies the behaviour of the *extractReports* method when there are redundancy elements only in the main part of USs
getTotalRedundancyElements (testExtractReports)	Provision of a CDA report for a US-pair that contains redundancy clauses in the "Main"and "Benefit" parts	Check whether the count of redundancy clauses in the main and benefit parts of the USs matches the value "Total Redundancy Clause" specified in the JSON_Report file	Verifies the behaviour of the *extractReports* method when there are redundancy elements in the main and benefit parts of USs
highlightPersona (testExtractReports)	Providing a CDA report for a US-pair with redundancy clause in "Triggers"edge (from Persona to Primary Action)	The persona should only be marked with hash symbol if there is a redundant clause in the main part	Checks the behaviour of the *extractReports* method when highlighting redundant personas in USs
BenefitInBothUss (testhigHlightRedundancies)	Provision of a CDA report for a US-pair where both USs have "benefit" part	Make textual report contains both USs with their respective "main" and "benefit" parts	Verifies the behaviour of the *highlightRedundancies* method when both USs have benefits
ContainInBenefitPart (testHighlightRedundancies)	Provide a CDA report for a US-pair with redundancy clauses of "Contains" within "Benefit" part	The entities included in Contains should be marked with hash symbol	Checks the behaviour of the *highlightRedundancies* method when highlighting redundant personas in USs

Table 2: Test cases for ReportExtractor class