Using Artificial Intelligence for the Specification of m-Health and e-Health Systems

Kevin Lano, Sobhan Y. Tehrani, Mohammad Umar and Lyan Alwakeel

Abstract Artificial intelligence (AI) techniques such as machine learning (ML) have wide application in medical informatics systems. In this chapter we employ AI techniques to assist in deriving software specifications of e-Health and m-Health systems from informal requirements statements. We use natural language processing (NLP), optical character recognition (OCR) and machine learning to identify required data and behaviour elements of systems from textual and graphical requirements documents. Heuristic rules are used to extract formal specification models of the systems from these documents. The extracted specifications can then be used as the starting point for automated software production using model-driven engineering (MDE). We illustrate the process using an example of a stroke recovery assistant app, and evaluate the techniques on several representative systems.

1 Introduction

Health informatics systems (*e-health systems*) include electronic health record management and analysis, diagnostic tools for clinicians, patient management systems for general practitioners and hospitals, and a wide range of mobile health (*m-health*) apps for general and specific health purposes. Health applications may need to deal with very large amounts of data, which may include uncertain information, and often involve multiple categories of users and stakeholders. The processed information is usually subject to high confidentiality requirements, eg., due to GDPR and related legislation.

Kevin Lano, Mohammad Umar, Lyan Alwakeel
Informatics Department, King's College London, UK e-mail: kevin.lano@kcl.ac.uk,e-mail: mohammad.umar@kcl.ac.uke-mail: lyan.alwakeel@kcl.ac.uk
Sobhan Y. Tehrani
Dept. of Computer Science, University of Roehampton, UK e-mail: sobhan.tehrani@roehampton.ac.uk

For each type of health application, software correctness is of high and even critical importance: incorrect patient information or analysis results can impair treatment or lead to incorrect and harmful medical interventions. Apps for patient self-management or health advice must also provide accurate and appropriate information (Public Health England, 2017). The well-publicised problems which arose in the UK in 2020 with the nationally-distributed COVID-19 advisory app highlighted the particular challenges of implementing m-health apps.

Therefore, rigorous requirements analysis is of key importance for e-health and m-health applications. In this chapter we address this challenge by providing assistance tools to automate the formalisation of software requirements statements, using natural language processing (NLP) and machine learning (ML) techniques. NLP and ML are instances of artificial intelligence (AI) techniques, which are increasingly being applied to support software engineering and development activities, in addition to widespread use in e-health and m-health applications, for example, to perform activity classification of mobile users or deducing calorie counts by image recognition applied to a photograph of a meal.

In an agile development context, parts of a system may be developed in different iterations, and their requirements are analysed using techniques such as interviews and exploratory prototyping in close collaboration with customer representatives. The techniques described in this chapter can be used to support the construction of models and prototypes from requirements, and enable customers to quickly see the semantic consequences of different requirements statements. We use the AgileUML toolset (Eclipse, 2021) to visualise and analyse formalised specifications. The toolset can be utilised to synthesise code in several 3GLs, including C, Java, Python and Swift. Mobile apps for the Android and iOS platforms can also be generated (Lano et al., 2021).

Section 2 describes the stroke recovery assistant case study and highlights some of the aspects of requirements statements which need to be addressed by a requirements formalisation approach.

Section 3 describes our requirements formalisation approach, using the case study to illustrate the process steps. In Sections 3.1, 3.2 and 3.3 we summarise the application of NLP, ML and OCR techniques in our process for automated requirements engineering (RE). In Sections 3.4 and 3.5 we describe specific techniques for the formalisation of data and behavioural requirements from natural language statements, and give examples from the case study.

Section 4 describes how automated RE can be used in an overall agile development process.

Section 5 surveys related work, and Section 6 gives conclusions.

2 Requirements Statement Example: The Stroke Recovery Assistant Case Study

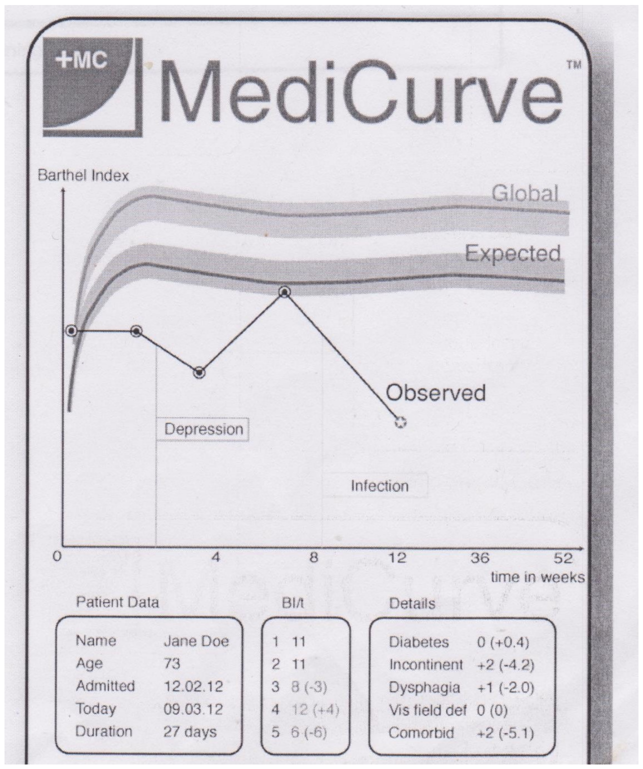
Requirements statements are typically written in natural language, and usually consist of a combination of text and diagrams, with text predominant. Requirements statements may include non-technical issues, such as the roles of stakeholders in the development process (Robertson and Robertson, 2012). In this chapter we only consider the technical requirements of the data and functionality of the system to be constructed.

We take as a typical example of a health application the stroke recovery assistant of (King's Health Partners, 2011). This involves both in-hospital systems for doctors and researchers, and a mobile app for patients to track their own progress. The requirements statement consists of:

- One page of text defining:
    1. the high-level goals of the project: "To improve the quality of life of patients who have suffered a stroke"; "To forecast recovery paths based on historical data and prediction algorithms"; "To build a data warehouse of clinical data".
    2. The scope of the system, including "A mobile app and web-based interface for data collection – for use by doctors in hospital"; "A mobile app and web-based interface for doctors to estimate recovery curves"; "A mobile app and web-based interface for patients to access information about recovery exercises, get information about their process, and track their recovery progress"; "An algorithm to compute recovery curves based on historical data".
    3. Security and infrastructure requirements, eg., the use of secure data connections, and the use of a single database for internal and external use.
    4. Some further objectives and success criteria, which are actually user stories, eg: "Allow doctors to collect a patient's data while they are in hospital".
- One page of informal diagrams showing the expected user interface behaviour of the doctor's UI, and a mockup of the mobile UI for doctors (Figure 1).
- One page of "supplementary requirements" which details the requirements from page 1, and number these as REQ-1 to REQ-10.

The document is typical of software requirements documents, in that:

- It is almost completely informal, written at a high level of abstraction.
- Substantial additional background knowledge and information must be obtained before development can begin – such as the precise definition of the recovery prediction algorithm to be encoded in software.
- Conflicts exist between different requirements, eg., the requirement to have a single database potentially conflicts with the requirements to provide both active data and research data: for confidentiality purposes the research data should be separate from active data and anonymised.
- Significant details are omitted, such as the design of the patient's UI, which will require specialised approaches to achieve usability for this group of patients.



Patient Data

Name	Jane Doe
Age	73
Admitted	12.02.12
Today	09.03.12
Duration	27 days

BI/t

1	11
2	11
3	8 (-3)
4	12 (+4)
5	6 (-6)

Details

Diabetes	0 (+0.4)
Incontinent	+2 (-4.2)
Dysphagia	+1 (-2.0)
Vis field def	0 (0)
Comorbid	+2 (-5.1)

Fig. 1 Mock-up of doctor's mobile UI. Source: Authors

Although a requirements engineer will take this requirements statement as their starting point, substantial additional investigation and requirements elicitation from the customer and using background documents is necessary. We will also use supplementary documentation (such as (Collin et al., 1988)) in our semi-automated RE process applied to this case study.

3 Requirements Formalisation Process and Techniques

There are typically four main stages in any requirements engineering (RE) process (Kotonya and Sommerville, 1996):

1. Domain analysis and requirements elicitation
2. Evaluation and negotiation
3. Requirements specification and formalisation
4. Requirements validation and verification.

A wide range of RE techniques can be used at these stages, such as document mining and structured and semi-structured interviews for requirements elicitation, and experimental prototyping and formal reviews for requirements validation.

In our approach, we focus on requirements *formalisation* to support the requirements specification of technical requirements, and do not address requirements elicitation. However, the process of formalisation may clarify the understanding of requirements by the customer and developer by providing a formal model and visualisation of requirements statements (eg., Figure 5 shows a visualisation of a formal model derived from the stroke recovery assistant requirements). The process could also contribute to requirements evaluation and validation by detecting potential conflicts, redundancy and omissions in the requirements statements, and by providing a unified technical specification for formal review.

As highlighted above, software requirements statements are typically expressed in a combination of natural language text and informal diagrams, with text as the predominant element. Thus we work with text as the input, which may include text derived from diagrams using OCR. The first stage of requirements formalisation involves extracting sentences from requirements documents and adding linguistic and semantic information as annotations to these sentences. In Sections 3.1, 3.2 and 3.3 we describe the application of NLP, ML and OCR for this stage.

Following the extraction of semantic information from requirements statements, the second stage of the formalisation process builds Unified Modelling Language (UML) and Object Constraint Language (OCL) models in two successive steps: extraction of a data model (Section 3.4) and extraction of a behavioural model (Section 3.5).

3.1 Natural Language Processing

NLP is a collection of techniques for the processing of natural language text, including:

- Part-of-speech (POS) tagging/classification: identifying the linguistic category of words in a text, such as nouns, adjectives, verbs, etc.
- Tokenisation and splitting of text into sentences.
- Lemmatisation: identifying the root forms of verbs, nouns, etc.
- Syntax analysis: construction of a parse tree identifying the subclauses of a sentence and their hierarchical relationships.
- Dependency analysis: identifying the roles of words in the sentence, ie., that certain nouns are the subjects and others the objects of the sentence, and that other words act as adjectival modifiers of nouns.
- Named entity recognition: identifying known terms such as cities, countries, etc.
- Reference correlation: identifying when the same element is referenced from different sentences.

NLP tools include Stanford NLP (Stanford University, 2020), Apache OpenNLP (Apache Software Foundation, 2019), iOS NLP Framework, and WordNet (Fellbaum, 2010).

The standard parts of speech include (Santorini, 1990):

- Determiners – tagged as *_DT*, eg., "a", "the".
- Nouns – *_NN* for singular nouns and *_NNS* for plural.
- Proper nouns – *_NNP* and *_NNPS*.
- Adjectives – *_JJ*, *_JJR* for relative adjectives, *_JJS* for superlatives.
- Possessives – *_PRP*$.
- Modal verbs – *_MD* such as "should", "must".
- Verbs – *_VB* for the base form of a verb, *_VBP* for present tense except 3rd person singular, *_VBZ* for present tense 3rd person singular, *_VBG* for gerunds, *_VBD* for past tense.
- Adverbs – *_RB*.
- Prepositions/subordinating conjunctions – *_IN*.

Figure 2 shows the metamodel of linguistic information which we use in automated RE. The information is obtained using NLP tagging, parsing and dependency analysis.

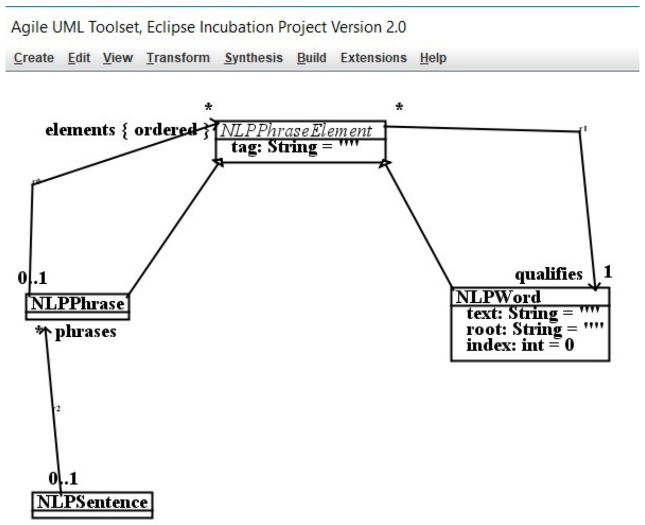


Fig. 2 NLP metamodel. Source: Authors

NLP techniques are relevant to the automation or semi-automation of requirements engineering activities such as extracting requirements from documentation, or formalisation of requirements as UML models, and they have been used in many works in these areas (Umar, 2020).

However the trained models available with the existing NLP tools are usually oriented towards general English text, which differs significantly from the subset of English typically used in software requirements statements. In particular, requirements statements do not usually use colloquial or casual English, and they use computing/software terminology. Words such as "track", "security" and "record" are used in a more specific way in software descriptions, compared to their use in general English. The existing models therefore sometimes misclassify words in requirements statements. For this reason we decided to re-train a POS tagging model using a corpus of requirements statements which we collected for data and behavioural requirements analysis (Sections 3.4, 3.5).

Table 1 shows the original POS tagging error rates (for words and for sentences) on the requirements documents used for training, and Table 2 the original and revised error rates on test cases after retraining of the Apache OpenNLP *maxent* POS model with manually-corrected versions of the tagging. In the training set 0.5% of words are misclassified, and 12.9% of sentences contain a misclassified word, which significantly affects the accuracy of our formalisation processes.

Retraining of the tagger reduced the error rates on the test cases from 0.4% per word and 9.4% per sentence, to 0.2% per word and 5.6% per sentence (Table 2). This process was effective in removing the more serious errors, however tagging and syntax analysis errors still arise in some cases (Section 4).

Case (# sentences)	*Tagging word errors*	*Tagging sentence errors*
Stroke assistant (8)	1	1
Trackformer (4)	2	2
g02 (99)	17	16
g03 (60)	11	7
g04 (51)	7	5
g05 (53)	11	10
g08 (66)	2	2
g10 (98)	15	12
g11 (74)	14	12
g12 (53)	3	2
g13 (53)	12	10
g14 (67)	15	13
g16 (66)	8	7
g17 (64)	9	8
g18 (101)	16	15
g19 (138)	26	24
g21 (73)	12	10
g22 (83)	8	5
g23 (56)	3	3
Error rates	0.5%	12.9%

Table 1 POS tagger error rates

Case (# sentences)	*Original tagging errors (words, sentences)*	*Retrained tagger errors (words, sentences)*
g24 (52)	9, 9	3, 3
g25 (100)	3, 3	3, 2
g26 (100)	13, 13	6, 6
g27 (115)	9, 7	4, 4
g28 (60)	9, 8	10, 10
Error rates	0.4%, 9.4%	0.2%, 5.6%

Table 2 Retrained POS tagger error rates

3.2 Machine Learning

Machine learning covers a wide range of techniques by which knowledge about patterns and relationships between data is gained and represented as implicit or explicit rules in a software system. ML can be used for classification of inputs (eg., to classify the severity of a patient's condition), translation (as in machine translation of natural language) or prediction. The techniques include K-nearest neighbours (KNN), decision trees, inductive logic programming (ILP) and neural nets. In each case there is typically a training phase, in which known relationships of existing data are provided to the ML software and rules expressing these relationships are induced, and a testing phase, to assess the accuracy of the learned rules on new data. The rules are usually embodied in an ML model file. Given a corpus of data with known classifications/results, this corpus is partitioned into a training set and a testing set, normally in a ratio such as 80:20. Only the training set is used to learn rules and construct an ML model. The accuracy of the trained ML model is typically assessed in terms of the level of agreement between its predictions on the test set, and the correct results.

Toolsets for ML include Google MLKit, Tensorflow, Keras, ScikitLearn and Theano.

NLP and ML can be usefully combined in requirements formalisation, whereby detailed linguistic information from NLP analysis can be provided as inputs to an ML process. Additionally, ML can be used to learn specific POS tagging rules for the restricted set of natural language texts that arise in software requirements documents, as described above.

3.3 Requirements Formalisation from Diagrams

Requirements statements often consist of a combination of text and informal diagrams, eg., (King's Health Partners, 2011; Robertson and Robertson, 2012). Apart from their visual content, such diagrams may also provide additional textual content, which can be used to identify key background terminology, data and behaviour (eg., as in Figure 1).

To extract the textual content of diagrams, we use optical character recognition (OCR). We applied the Google MLKit OCR library to example diagrams including those of (King's Health Partners, 2011; Robertson and Robertson, 2012) and the TS33.102 security standard of 3GPP[1]. While the basic OCR algorithm is able to recognise individual typed words, it is not effective at recognising blocks of text spread over successive lines. For example, 59 of 60 individual words in the road maintenance system context diagram of (Robertson and Robertson, 2012) are correctly recognised, but only 5 out of 19 blocks of text. To address this issue, we extend the OCR algorithm to consider two text elements as part of a single block if they are (approximately) left-aligned or right-aligned and are vertically immediately adjacent. Text blocks are merged by this process until no further combinations are possible. This increases the number of correct blocks and removes incomplete and incorrect blocks.

Accuracy in this chapter is measured by the standard *F-measure* defined by $F = \frac{2*p*r}{p+r}$ where precision $p = \frac{\textit{correct identifications}}{\textit{total identified}}$ and recall $r = \frac{\textit{correct identifications}}{\textit{total correct}}$. Table 3 shows the F-measure for individual word recognition and for the original and enhanced text block recognition algorithms. A limitation of this approach is that the basic OCR algorithm sometimes identifies incorrect bounds for text areas, which is a factor in case 4 of Table 3.

Case	*F-measure for word detection*	*F-measure of block detection*	*F-measure for merged block detection*
1: Road maintenance (Robertson and Robertson, 2012)	0.97	0.38	0.65
2: Stroke assistant (King's Health Partners, 2011)	0.93	0.59	0.83
3: 3GPP TS33.102 (1)	0.9	0.64	0.88
4: 3GPP TS33.102 (2)	0.91	0.41	0.51

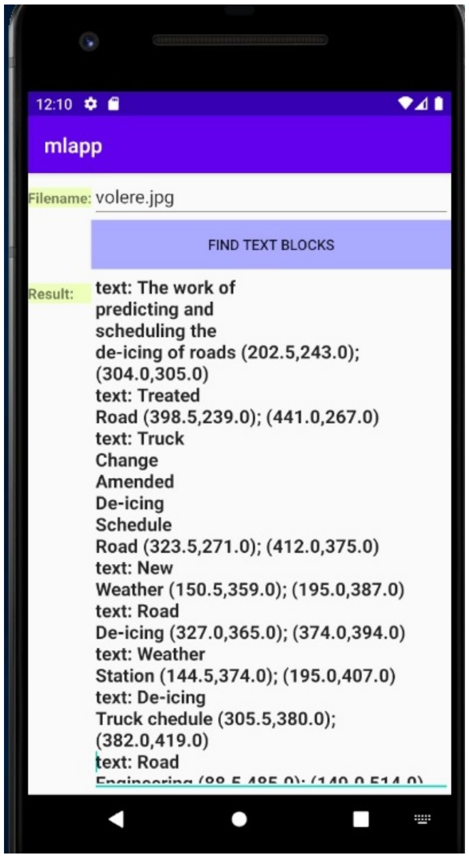
Table 3 Original and enhanced text recognition accuracy

Object recognition and OCR need to be coordinated, and text elements derived from a sketch image associated with other visual elements from the image (eg., class rectangles or association lines) if they satisfy relevant spatial conditions (containment/overlapping or proximity constraints). We provide our enhanced OCR analysis as an Android app (Figure 3). The app displays the merged text blocks, and also records these in a file.

3.4 Deriving Data Model Specifications from Requirements Statements

The data model of an application is the basis for functional and behavioural specifications, thus we formalise this model prior to formalising functionality. As input to

[1] https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2262. Retrieved May 15, 2021



volere.jpg

```
text: The work of
predicting and
scheduling the
de-icing of roads (202.5,243.0);
(304.0,305.0)
text: Treated
Road (398.5,239.0); (441.0,267.0)
text: Truck
Change
Amended
De-icing
Schedule
Road (323.5,271.0); (412.0,375.0)
text: New
Weather (150.5,359.0); (195.0,387.0)
text: Road
De-icing (327.0,365.0); (374.0,394.0)
text: Weather
Station (144.5,374.0); (195.0,407.0)
text: De-icing
Truck chedule (305.5,380.0);
(382.0,419.0)
text: Road
```

Fig. 3 Extended OCR app. Source: Authors

this step we use the explicit requirements statements of the app, and any necessary background documents and pre-existing models. In the medical domain there may already exist detailed specifications of particular hardware to be used. For example, the diffusion pump controller (Campos and Harrison, 2011) is of this kind. Particular medical models may also be assumed, such as the Barthel Index model (Collin et al., 1988) for the performance of daily living activities (ADL), in the case of the stroke recovery assistant app. Our approach accommodates this situation by integrating the requirements formalisation approach with the model editor of AgileUML, so that any prior existing model can be loaded and its elements searched to identify and classify named terms in the analysed requirements statement. We assume that data and behaviour requirements are clearly separated, with global behaviour requirements expressed as user stories.

To assist in statement classification and semantic analysis, we define a knowledge base of software requirements terminology, including parts-of-speech (POS) information on words, synonyms, and semantic properties of terms (eg., that "temperature" is likely to denote a real-valued attribute).

For specific domains such as medicine or telecoms, a domain-specific knowledge base is also necessary, to capture information on terms with a specialised meaning in the domain, such as "dysphagia", "intervention", "consultation" and "out-patient" in medicine, and "connection" and "cell" in telecoms. A domain-specific knowledge base will usually take precedence over the general knowledge base.

We use the Stanford NLP toolset (Stanford University, 2020) and Apache OpenNLP (Apache Software Foundation, 2019) to perform part-of-speech tagging of English sentences, to construct syntax trees, and to identify dependencies within sentences. The resulting information is then used as input for a classifier which identifies sentences as either:

1. Definitions of classes
2. Definitions of specialisation/generalisation relations between classes
3. Definitions of associations
4. Definitions of invariants or other properties of classes.

After classification, a detailed syntactic and semantic analysis can be applied to extract model content from the text information. Successive sentences are analysed taking account of the partial models already obtained from preceding sentences, so that models are progressively elaborated. Contradictions, redundancy and omissions within requirements statements can also be identified during this process.

The wide variability in expression of data requirements statements in practice can be seen in Table 4.

Statement type	*Examples*
Class definition	"Each bond has a unique name, a numeric term, a numeric coupon, and whole-number frequency"
	"The app records the weight, age, height and sex of users"
	"The property table holds details of address, price and vendor"
Specialisation/ generalisation	"A bond is either a corporate bond or government bond"
	"A customer is a special case of a user"
Associations	"Each Bond is issued by one issuer"
	"The app records a training programme for each user"
	"Each student attends several courses"
Invariants	"Bond durations are never more than the bond term"
	"The sex is either male or female"

Table 4 Variation in requirements statement sentences

In order to classify sentences as one of the above four categories, we apply heuristic rules based on a number of factors: 1) the similarity of the main verb to one characteristic of the classification category; 2) occurrences of terminology indicating the classification category; 3) sentence structure and intra-sentence dependencies; 4) classifications of terms from the general knowledge base.

Characteristic features of class definition sentences are:

- Occurrence of verbs synonymous to "have", "has", "holds", "consists", "specifies", "comprises", "is defined by", "stores", "records", "maintains", etc
- The class being introduced or further defined is the main noun of the sentence subject, except in cases where the subject denotes the system/app, in which case the class occurs in the object part
- Terminology such as "each/every X instance ..."/"all X instances ..."/"An X instance" suggesting that X is a class
- Terms denoting attributes occur in the object part.

Characteristics of specialisation/generalisation sentences are:

- Occurrence of verbs synonymous to "specialises", "generalises", "inherits", "extends", "abstracts", "classified", "derived", etc, or the sentence structure is similar to forms such as "... is a special case of ..."; "... is either an X or ..."; "... is an X [with ...]"
- Adjectival qualifiers applied to already-known classes, suggesting that the qualified term denotes a subclass of the unqualified: "secure channel"
- A conjunctive/disjunctive clause combines two or more known class terms: "patients and caretakers"; "treatments and exercises"
- Usually the only named elements are classes, not features
- If the left hand side (subject) does denote a known feature, then the sentence is instead a constraint defining the permitted values of the feature.

Characteristics of association definition sentences are:

- Occurrence of verbs synonymous to "linked to"/"related to"/"associated with", etc, or the use of an attribution verb ("has", "composed of", etc) with the object naming classes.
- Alternatively, the main verb may name the association role, with the related class *Y* in the sentence object: "Each X role [quantifier] Y(s)"
- Usually the only named elements are role names and classes, not attributes.

Invariants are difficult to characterise, but the occurrence of words synonymous to "is always"/"are never" or of comparatives/superlatives (*_JJR*, *_JJS*) is one indicator of this form. The subject may be a feature f of class X in a possessive phrase of the form "The X's f" or equivalent. Comparator and mathematical operators may be used.

In order to detect synonyms, we use a thesaurus and word distance measures of name edit distance (NSS) (Levenshtein et al., 1966) and name semantic similarity (NMS) (Lano et al., 2020). Users can set a threshold for the level of similarity which is to be regarded as significant. We found that the WordNet thesaurus (Fellbaum,

2010) was too general for practical use, since it covers all possible meanings of a word in general English, whilst requirements statements use a restricted and usually technical vocabulary. Thus we developed our own thesaurus.

We use the standard XML thesaurus format of (Rowley, 2007) to define this thesaurus, and to represent domain and general background knowledge in knowledge bases. An example from the general requirements knowledge base is:

```
<CONCEPT>
<DESCRIPTOR>temperature</DESCRIPTOR>
<POS>NN</POS>
<SEM type="double">attribute</SEM>
<PT>temperature</PT>
<NT>heat</NT>
<NT>body-temperature</NT>
<NT>ambient-temperature</NT>
</CONCEPT>
```

Any preferred term (*PT*) or non-preferred term (*NT*) in the concept word group will be treated as having the same semantics and part-of-speech classification defined in the *SEM* and *POS* clauses. Care should be taken to avoid ambiguity and conflicting definitions of the same word when designing or extending a knowledge base of this form.

Subsequent to classification, detailed semantic analysis takes place in three phases applied over the sequence of sentences in the requirements statement:

1. Recognition of classes:
    a. Words identical to the names of classes already existing in the software model or recognised from a previous sentence;
    b. Words denote classes according to background knowledge in the general or specialised knowledge base (eg, “webpage”, “ward”);
    c. Words denote classes according to their role in the sentence (eg., as the subject of the principal form of class definition sentence);
    d. Words are proper nouns (tagged as NNP or NNPS) – the singular form of the noun is taken as the class name. A more liberal approach could be to also consider other nouns (NN or NNS) with an initial capital as proper nouns, if they do not appear as the first word of a sentence.

    Inheritance relations are established based on the sentences classified as defining specialisations.
2. Recognition of attributes:
    a. A word is identical to the name of an attribute already known from the pre-existing software model or from previous sentences;
    b. A word denotes an attribute according to general or specific background knowledge (eg., “age”, “dose”);
    c. A word is an attribute according to the role of the word in the sentence (eg., it occurs in the object of class definition sentences, and it is not a class or role).

The types, multiplicities and stereotypes of attributes are also extracted.

3. Reference recognition:

   a. A word is identical to the name of a reference already known from the pre-existing software model or from previous sentences;
   b. A word denotes a reference according to background knowledge (eg., "parent");
   c. The word is a reference according to the form of the sentence (eg., "The monitoring equipment includes a set of movement sensors");
   d. The word has its initial letter in lower case, and the initial uppercased singular form of the word is the name of a known class (eg, "patients", "ward").

   Reference types, multiplicities and stereotypes are also extracted (eg., "series" suggests an $\{ordered\}$ stereotype on a feature).

In order to validate the data formalisation approach, we compared the result of formalisation with models manually created from the same requirements statements by a modelling expert. We used 11 cases, including (Marconi Command and Control Systems, 1990), (King's Health Partners, 2011) and three cases from (Kaggle, 2021). These contain 67 sentences in total. We achieved an average F measure of 0.91 for the accuracy of sentence classification, and 0.84 for the accuracy of models.

The formalisation process can also assist in requirements validation, by detecting redundancy, omissions or inconsistencies in requirements statements (Table 5).

Redundancies	*Inconsistencies*	*Omissions*
Two statements both define feature f of class C.	Two statements give conflicting types/multiplicities for feature f of C.	A class is referenced but not defined.
Two statements both express that class D subclasses class C.	Two statements express contradictory subclass relationships between classes.	A subclass relation is assumed but not declared.

Table 5 Redundancy/conflicts in data requirements statement sentences

Possible quality problems with the synthesised model can be identified, such as 1-1 associations and multiple inheritance. Some potential cases of refactoring can also be identified, such as multiple subclasses of the same class all possessing copies of the same feature. Classes which are synonymous or have identical data can be merged.

Traceability of the specification with respect to the requirements is maintained by recording a many-many relation $dependsUpon : ModelElement \leftrightarrow NLPPhraseElement$ which identifies which requirements statement elements contributed to the definition of each derived model element. The originating elements $dependsUpon(\!|\ \{x\}\ |\!)$ of a class or use case x (for behaviour analysis) are recorded as stereotypes of x.

3.5 Deriving Behavioural Model Specifications from Requirements Statements

Following data formalisation, behavioural formalisation can be applied, this uses the extracted data specification.

We assume that behavioural requirements are of two kinds:

- Expected functionalities/services offered by the system, expressed as user stories. These can be formalised as UML use cases in AgileUML.
- Operation specifications for operations/methods of particular classes, expressed in terms of the expected inputs and outputs of these operations.

In this chapter we focus upon the first case.

User stories are the principal form of behaviour requirement used in agile methods. They express some unit of functionality which a user of the system expects from the system, in other words, the use cases which the system should support. User stories are typically expressed in the format

[*actor identification*] *goal* [*justification*]

where the actor identification and justification are optional.

For example:

"As an A, I wish to B, in order to C"

The actor identification defines which system actor the use case is for. The goal describes the intended use case actions, and the justification explains its purpose. These correspond to the graphical use case form of Figure 4.

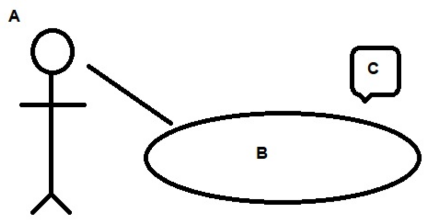


Fig. 4 Use case derivation from user stories. Source: Authors

Other alternative formats for user stories are

"The A should be able to B, in order to C"
"The A must be able to B, so that C"

"The system must allow A to B"
"The system shall provide B"
"A will be able to B"

or simply *B* by itself, as in the (King's Health Partners, 2011) patient analysis behavioural requirements for the doctor's interface:

```
Add new user.

Add new patient.

Add new patient data.

Update existing patient data.

Patient-specific numerics.

Patient-specific graphics.

Group-level graphics.

Group-level numerics.
```

These were extracted from an informal diagram of the system behaviour using the Google MLKit OCR facilities.

In each format, a reference to any actor *A* occurs prior to *B*. A modal verb ("should", "must", etc) identifies the obligation to provide the functionality *B*. Statements *B* and *C* typically refer to the classes and features recognised in the data formalisation stage. *B* should begin with a verb in infinitive form (the tag *VB*). In the absence of a formalised data model, classes are recognised according to background knowledge and their role in the sentence. For example, *User*, *Patient* and *Group* are recognised as classes in the above patient analysis requirements.

In addition to the background knowledge file *output/background.txt* on nouns, we also provide a knowledge base *output/verbs.txt* of verbs and their classifications (*read*, *edit*, *create*, *delete*, *other*). This file is in the same thesaurus format as the background file for nouns, and can be edited by the user to provide further information on verbs. An example entry is:

```
<CONCEPT>
<DESCRIPTOR>search</DESCRIPTOR>
<POS>VB</POS>
<SEM>read</SEM>
<PT>search</PT>
<NT>find</NT>
<NT>retrieve</NT>
<NT>extract</NT>
<NT>scan</NT>
```

```
<NT>inspect</NT>
<NT>identify</NT>
<NT>examine</NT>
</CONCEPT>
```

User stories that represent standard functionalities such as creating, editing, deleting and reading instances of a class, can be classified into these categories based on the verbs used in the goal part *B* of the user story. The classified sentence is then formalised as a UML use case. The semantics of the use case depends on the classification of the user story. The actor *A* of a use case is recorded in a stereotype *actor* = "*A*" of the formalised use case. If *A* is a class name, then an instance of the actor becomes the first parameter of the use case.

Data dependencies between classes and use cases can then be computed and shown visually (eg., Figure 5).

We validated the behaviour formalisation process using the 22 requirements statements of (Mendeley software repository, 2018), 3 cases from (Kaggle, 2021) and the stroke recovery assistant case. In total there are 2218 user stories in these cases. The accuracy of use case classification and the accuracy of the formalised UML models was measured. In computing the F-measure for formalisation we consider how many synthesised classes and use cases are semantically valid or invalid wrt the requirements. The overall average for classification accuracy was 0.97, and for formalisation 0.94.

The main source of imprecision in behaviour formalisation is due to errors in tagging, eg., tagging of "And" as a *NNP* instead of *CC*. In addition, (i) general concepts such as "Application", "UI" or "System" are incorrectly recognised as classes; (ii) successive nouns in the actor identification part are not combined, but become separate classes, eg., "Attending Physician" is represented as two separate classes.

These issues could be corrected by (i) defining a separate list of keywords/blocked words, which cannot be used as class names; (ii) automatically combining successive nouns in the actor identification.

We prefer to leave (ii) as a developer choice, so that if they want the combined term to become a class, they can write it as one word "AttendingPhysician", etc.

As with the formalisation of data requirements, possible flaws in user stories can be detected during this analysis process:

- The goal part *B* does not begin with a verb;
- The formalised use case has multiple possible classifications (eg., create and delete), indicating a user story that is too complex and should be decomposed;
- A complete requirements statement would be expected to have a *createE*, *editE* or *readE* use case for each entity *E* of the system.

Duplicated requirements sentences and multiple variant versions of the same concept can also be detected by the user story formalisation process. A useful facility which could be added would be to group those use cases with a common actor into separate subsystems. Likewise it would be useful to be able to group together all use cases that

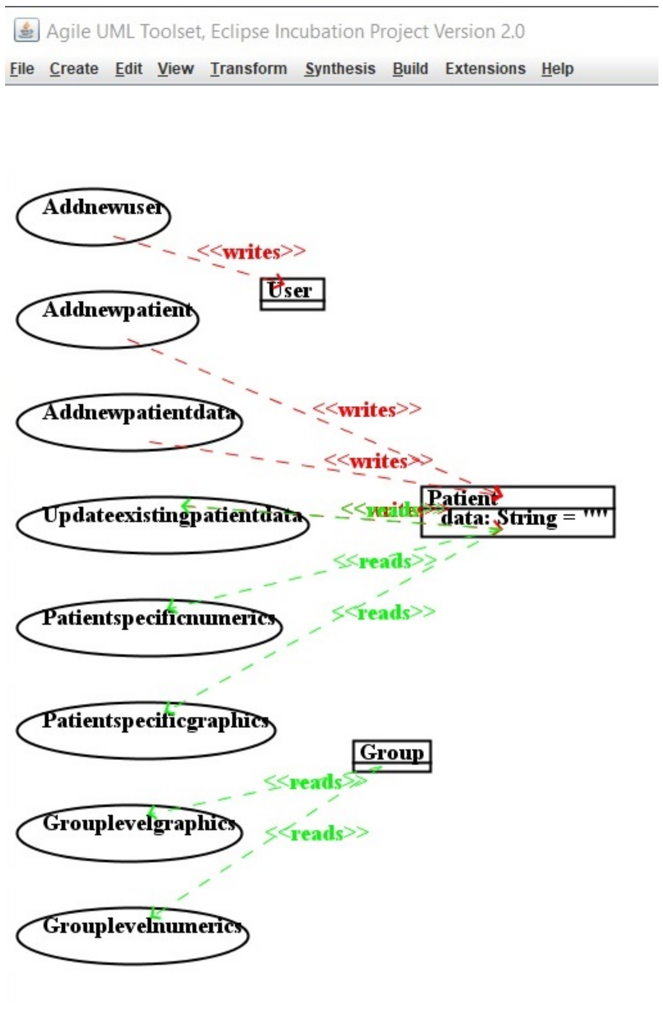


Fig. 5 Formalised model and dependencies of (King's Health Partners, 2011) patient analysis requirements. Source: Authors

refer to a given class or subset of classes. The formalised models contain sufficient information to perform such modularisations.

4 Integration of Automated Requirements Engineering into an Agile MDE Process

In an agile method such as Scrum or XP, parts/subsystems of a system are developed and delivered within iterations or "sprints". Lifecycle stages of feasibility analysis, requirements capture and requirements specification are performed for each system part, and themselves may involve iteration. For example, a prototype may be constructed to demonstrate how the developer intends to implement the subsystem requirements. This can then be reviewed by the customer representative in order to check that the intended approach satisfies the customer requirements. Modifications are made if necessary and the prototype progressed towards a production version.

The automated RE techniques described in Sections 3.4, 3.5 can accelerate this development iteration cycle, by automating model construction and the synthesis of prototypes from these models (Figure 6). Traceability is a key facility in this situation, enabling the developer and customer to identify which requirements statements contribute to possibly erroneous use cases and executable behaviour.

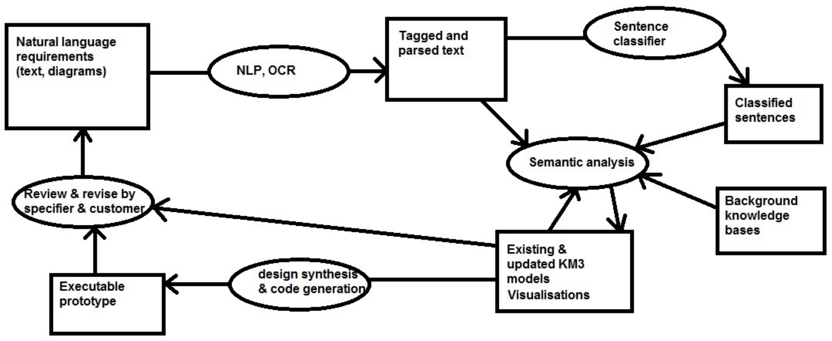

Natural language requirements (text, diagrams)
NLP, OCR
Tagged and parsed text
Sentence classifier
Classified sentences
Semantic analysis
Background knowledge bases
Review & revise by specifier & customer
Existing & updated KM3 models Visualisations
design synthesis & code generation
Executable prototype

Fig. 6 Automated requirements engineering in the agile MDE process. Source: Authors

The requirements of each system part may consist of a group of related user stories. These form a work item which will be listed in the product backlog of the system, and in the iteration backlog of the iteration responsible for its development. In an agile MDE process, a common class diagram model is usually shared by all the team working in an iteration, in order to ensure a consistent data model is used (Lano, 2017). The different groups of use cases will typically form separate work items, but each group depends upon the common data model. Together with the models and requirements statements, a project could also maintain a project-specific knowledge base, which records particular terminologies and word classifications relevant to the project.

As an example of this process, we consider some further data and behaviour requirements from (King's Health Partners, 2011):

```
For each patient,
record the BarthelIndex.

Each patient has a name, age,
date admitted, and stay duration.

Each patient has measures of diabetes,
incontinence, dysphagia,
visual field and comorbidity.

Each patient has estimates of mortality
risk and disability risk.

Patients are either inpatients or outpatients.

Comorbidity is either present or absent or
unknown.

As a nurse, I need to process
the admission of a patient to a
ward.

As a doctor, I wish to assess
the condition of a patient.

As a doctor, I need to approve
the discharge of a patient from
hospital.
```

Part of the formalised model extracted from the above sentences is:

```
package app {
  class BarthelIndex {
  stereotype originator="1";
  }

  class Patient {
  stereotype originator="1";
  stereotype modifiedBy="1";
  stereotype modifiedBy="2";
  stereotype modifiedBy="3";
  stereotype modifiedBy="4";

    attribute name : String;
    attribute age : double;
    attribute date : String;
    attribute duration : double;
    attribute measures : Set(String);
    attribute diabetes : String;
    attribute incontinence : String;
    attribute dysphagia : String;
    attribute visualField : String;
    attribute comorbidity : ComorbidityTYPE;
    attribute estimates : Set(String);
    attribute mortalityRisk : double;
    attribute disabilityRisk : double;
    reference record : BarthelIndex;
  }
```

Review of this model reveals several flaws: “admitted” and “stay” are omitted from the model, because they were misclassified by the NLP tagger as verbs. The parse tree of the second sentence is:

```
(ROOT
  (S
    (NP (DT Each) (NN patient))
    (VP (VBZ has)
      (NP
        (NP (DT a) (NN name)
          (, ,) (NN age)
          (, ,) (NN date))
        (SBAR
          (S
            (VP
              (VP (VBD admitted))
              (, ,)
              (CC and)
              (VP (VB stay)
                (NP (NN duration))))))))
    (. .)))
```

To avoid this problem, we can change the text to “Each patient has a name, age, admission date and stay duration”.

In addition, spurious attributes *measures* and *estimates* have been derived. This can be remedied by clarifying the third requirements sentence to:

```
Each patient has a diabetes measure,
incontinence measure, dysphagia measure,
visual field measure and comorbidity.
```

and similarly for the fourth sentence. The revised model derived from these improved requirements is shown in Figure 7.

A quality check of this model gives the warning that all subclasses of *Patient* are empty, so these could be refactored into an enumeration. Again, the requirements

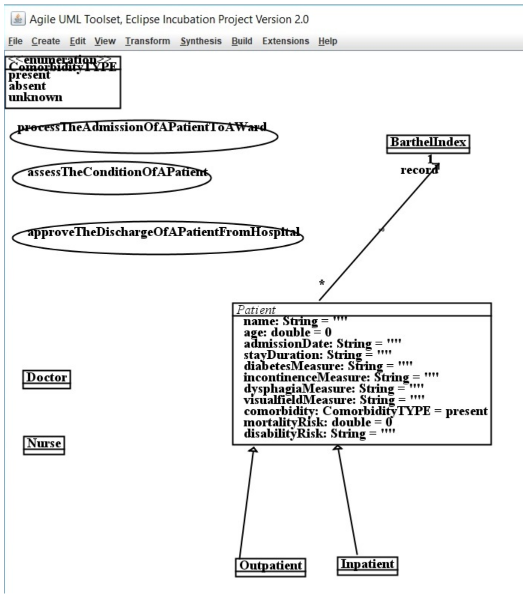

Agile UML Toolset, Eclipse Incubation Project Version 2.0
File Create Edit View Transform Synthesis Build Extensions Help
<<enumeration>>
ComorbidityTYPE
present
absent
unknown
processTheAdmissionOfAPatientToAWard
assessTheConditionOfAPatient
approveTheDischargeOfAPatientFromHospital
BarthelIndex
1
record
*
Patient
name: String = ""
age: double = 0
admissionDate: String = ""
stayDuration: String = ""
diabetesMeasure: String = ""
incontinenceMeasure: String = ""
dysphagiaMeasure: String = ""
visualfieldMeasure: String = ""
comorbidity: ComorbidityTYPE = present
mortalityRisk: double = 0
disabilityRisk: String = ""
Doctor
Nurse
Outpatient
Inpatient

Fig. 7 Formalised requirements of stroke assistant (patient analysis and management). Source: Authors

could be revised (*stayDuration* could be moved to *Inpatient*, for example). From this model Java code can be generated for a prototype app implementation (Lano et al., 2021), together with a test suite and test harness (Figure 8).

5 Related Work

In (Alwakeel and Lano, 2021) we evaluated 110 m-health apps in the domain of patient self-managed health-care. This survey identified that there has been increasing use of ML techniques in such apps, to perform activity detection and behaviour analysis. However this evaluation also identified that there is a lack of guidelines for the specification and design of such systems, and in particular for the selection of ML techniques. An MDE-based approach would therefore appear to be beneficial in increasing the rigour of m-health app development.

We also carried out a systematic literature review (SLR) of papers in the field of automated requirements engineering, published between 1996 and 2020 (Umar, 2020). From an original set of 3853 papers, 54 studies were short-listed and analysed in detail. We found that NLP was the main approach used, with 52% of cases using some NLP technique, and most approaches were semi-automated rather than fully-automated. Of NLP techniques, 13 cases used parsing, 11 used POS and 3 used dependency analysis. 14 cases used a combination of NLP and ML, with naive

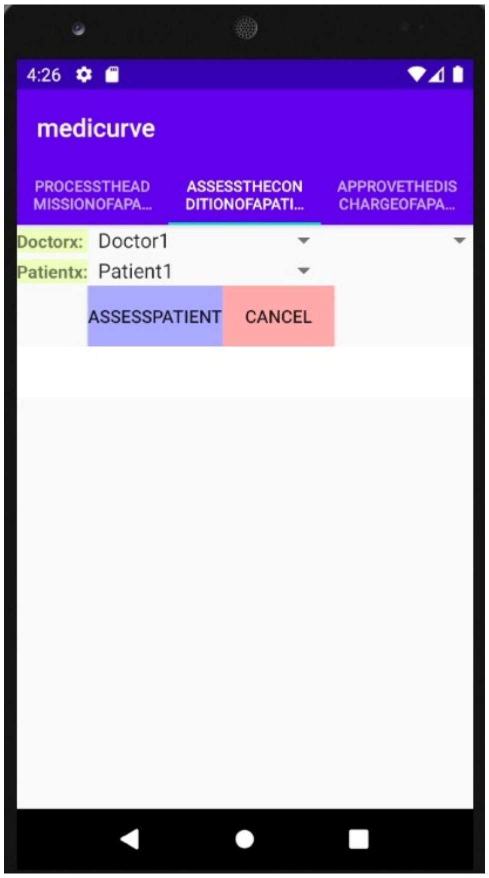

4:26
medicurve
PROCESSTHEAD MISSIONOFAPA...
ASSESSTHECON DITIONOFAPATI...
APPROVETHEDIS CHARGEOFAPA...
Doctorx: Doctor1
Patientx: Patient1
ASSESSPATIENT
CANCEL

Fig. 8 Stroke assistant app prototype. Source: Authors

Bayes the most common classifier used (4 cases) and neural nets were used in 2 cases.

Subsequent to the SLR, further relevant papers (Saini et al., 2020; Burgueño et al., 2019; Xu et al., 2020) have been published. Compared to (Saini et al., 2020) we also use a hybrid NLP/ML approach, and a knowledge base, however we adopt the use of both a general knowledge base and specialised knowledge bases for particular domains (finance and telecoms). Our approach is therefore closer to the framework for using NLP in requirements engineering suggested in (Berzins et al., 2007). With regard to datasets, (Saini et al., 2020) use a set of student solutions to a particular coursework problem, whilst we use diverse examples both from student projects and industrial cases, including the large dataset (Mendeley software repository, 2018) of user stories. While our accuracy measures are generally lower than those of (Saini et al., 2020), this may be due to the higher variability in the form of the requirements in our dataset.

In (Xu et al., 2020), a customised NLP approach is applied to formalise statements regarding physical construction layouts from textual requirements, and to express these in an extended OCL. The approach could have more general potential for formalising application invariants or standards documentation in OCL, however our preference is for the use of established NLP tools where possible, with customisation of the language models via retraining to specialised domains.

NLP techniques also have relevance for text mining tasks such as extraction of information from medical reports, to support analysis and prediction processes. In general, text mining is more effectively performed upon more highly structured and semantically coded documentation, rather than upon less structured source material.

National guidelines for health apps have been published in the UK (Public Health England, 2017). These emphasise the critical importance of the correctness in health apps. Our iterative approach can help to assure correctness by providing a resolution of ambiguities and errors in software requirements at an early development stage.

6 Conclusions

In summary, our requirements formalisation approach for health applications provides an effective means for extracting specifications from natural language requirements. Integration with an existing MDE toolset provides a direct means of visualising, analysing and prototyping formalised requirements.

In future work, we aim to:

- Extend the data and behaviour requirements formalisation techniques to encompass a wider range of input texts and diagrams, in particular to formalise detailed constraints and operation definitions;
- Investigate the production of other models from formalised requirements, such as activity models/workflows, identify use case relationships, and implement further quality checks/analysis on formalised requirements;
- Investigate the formalisation of models from sketches.

Acknowledgements Lyan Alwakeel acknowledges the funding support of University of Tabuk, Saudi Arabia.

Muhammad Aminu Umar acknowledges the funding support of the Petroleum Technology Development Fund (PTDF), the Federal Government of Nigeria.

The authors declare no potential conflicts of interest.

References

Alwakeel, L. and Lano, K. (2021). Functional and technical aspects of mobile health applications: A systematic literature review, King's College London.

Apache Software Foundation (2019). Apache OpenNLP Toolkit. https://opennlp.apache.org/.

Berzins, V., Martell, C., and Adams, P. e. a. (2007). Innovations in natural language document processing for requirements engineering. In *Monterey Workshop*, pages 125–146. Springer.

Burgueño, L., Cabot, J., and Gérard, S. (2019). An LSTM-based neural network architecture for model transformations. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 294–299. IEEE.

Campos, J. C. and Harrison, M. (2011). Modelling and analysing the interactive behaviour of an infusion pump. *Electronic Communications of the EASST*, 45.

Collin, C., Wade, D., Davies, S., and Horne, V. (1988). The barthel adl index: a reliability study. *International disability studies*, 10(2):61–63.

Eclipse (2021). Eclipse AgileUML project. https://projects.eclipse.org/projects/modeling.agileuml.

Fellbaum, C. (2010). WordNet. In *Theory and applications of ontology: computer applications*, pages 231–243. Springer.

Kaggle (2021). Software requirements dataset. www.kaggle.com/iamsouvik/software-requirements-dataset.

King's Health Partners (2011). Stroke recovery assistant project brief, version 1.0.

Kotonya, G. and Sommerville, I. (1996). Requirements engineering with viewpoints. *Software Engineering Journal*, 11(1):5–18.

Lano, K. (2017). *Agile model-based development using UML-RSDS*. CRC Press.

Lano, K., Alwakeel, L., Rahimi, S. K., and Haughton, H. (2021). Synthesis of mobile applications using AgileUML. In *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*, pages 1–10.

Lano, K., Fang, S., Umar, M., and Yassipour-Tehrani, S. (2020). Enhancing model transformation synthesis using natural language processing. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–10.

Levenshtein, V. I. et al. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union.

Marconi Command and Control Systems (1990). A requirement specification for a simple track former.

Mendeley software repository (2018). Mendeley user story dataset. accessed January 5th, 2021, https://data.mendeley.com/dataset/7zbk8zsd8y/1.

Public Health England (2017). Criteria for health app assessment. accessed May 20th, 2021, https://www.gov.uk/government/publications/health-app-assessment-criteria.

Robertson, S. and Robertson, J. (2012). Volere requirements specification template.

Rowley, J. (2007). Bs 8723 structured vocabularies for information retrieval: Part 1: Definitions, symbols and abbreviations, and part 2: Thesauri. *Journal of Documentation*.

Saini, R., Mussbacher, G., Guo, J. L., and Kienzle, J. (2020). DoMoBOT: a bot for automated and interactive domain modelling. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–10.

Santorini, B. (1990). Part-of-speech tagging guidelines for the Penn Treebank Project.

Stanford University (2020). Stanford nlp. accessed May 20th, 2021, https://nlp.stanford.edu/software.

Umar, M. A. (2020). Automated Requirements Engineering Framework for Agile Development. *ICSEA 2020*, page 157.

Xu, X., Chen, K., and Cai, H. (2020). Automating Utility Permitting within Highway Right-of-Way via a Generic UML/OCL Model and Natural Language Processing. *Journal of Construction Engineering and Management*, 146(12):04020135.