






This item is the archived peer-reviewed author-version of:

Expanding Normalized Systems from textual domain descriptions using TEMOS

Reference:

Šenkýř David, Suchánek Marek, Kroha Petr, Mannaert Herwig, Pergl Robert.- Expanding Normalized Systems from textual domain descriptions using TEMOS
Journal of intelligent information systems - ISSN 1573-7675 - 59:2(2022), p. 391-414
Full text (Publisher's DOI): <https://doi.org/10.1007/S10844-022-00706-8>
To cite this reference: <https://hdl.handle.net/10067/1881900151162165141>

Expanding Normalized Systems from Textual Domain Descriptions using TEMOS

David Šenkýř  · Marek Suchánek  ·
Petr Kroha  · Herwig Mannaert  ·
Robert Pergl 

the date of receipt and acceptance should be inserted later

Abstract Functional requirements on a software system are traditionally captured as text that describes the expected functionality in the domain of a real-world system. Natural language processing methods allow us to extract the knowledge from such requirements and transform it, e.g., into a model. Moreover, these methods can improve the quality of the requirements, which usually suffer from ambiguity, incompleteness, and inconsistency. This paper presents a novel approach to using natural language processing. We use the method of grammatical inspection to find specific patterns in the description of functional requirement specifications (written in English). Then, we transform the requirements into a model of Normalized Systems elements. This may realize a possible component of the eagerly awaited text-to-software pipeline. The input of this method is represented by textual requirements. Its output is a running prototype of an information system created using Normalized Systems (NS) techniques. Therefore, the system is ready to accept further enhancements, e.g., custom code fragments, in an evolvable manner ensured by compliance with the NS principles. A demonstration of pipeline implementation is also included in this paper. The text processing part of our pipeline extends the existing pipeline implemented in our system TEMOS, where we propose and implement methods of checking the quality of textual requirements concerning ambiguity, incompleteness, and inconsistency.

Keywords normalized systems, requirements engineering, natural language processing, model-driven development, code generation

D. Šenkýř · M. Suchánek · P. Kroha · R. Pergl
Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, 16000
Praha 6, Prague, Czech Republic
E-mail: david.senkyr@fit.cvut.cz

H. Mannaert · M. Suchánek
Faculty of Business and Economics, University of Antwerp, Prinsstraat 13, 2000 Antwerpen,
Antwerp, Belgium

1 Introduction

Software development always starts with analysis and requirement specifications to obtain a product requirement document describing what the software system must, should, or can do to fulfil its purpose, e.g., efficiently supporting specific activities of people and organizations. This document is then used according to the methodologies and development processes used to design, produce, test, and deploy the target software system. The development is typically supported by models. Therefore, models assembled from a requirement specification can be used to generate parts of the resulting system with applied best practices using model-driven development techniques [2, 9].

One such technique is based on Normalized Systems (NS) theory [11]. It focuses on the evolvability of software systems by using code templates to compose a system from building blocks called NS elements. A model of the NS elements together with various settings and configurations is used by NS expanders to fill in the templates and produce an NS application – typically an enterprise information system (EIS). Custom code fragments can be added to the generated code base; however, the code can still be regenerated (e.g., when the templates support model changes or new technologies) as custom code is harvested and then reinjected into the regenerated code. The NS approach has been verified in practice by numerous large-scale and long-term projects. [13, 14]

Currently, an NS model must be created manually by analysts based on textual requirement specifications or other domain knowledge, mostly captured as text. This paper aims to automatically build NS models from textual domain descriptions in natural language (English) to enhance this part of the NS development process. The goal is to create a method and a tool that will produce an initial NS model for further refinement by applying natural language processing (NLP). Together with existing tools supporting the NS development process, it should be possible to streamline the generation of an NS application prototype directly from text. We foresee several benefits, such as quick prototyping, including testing, making rapid and cheap changes, and, importantly, validating the original requirements. We follow the idea of symbiosis between the textual requirements and the developed system. In the end, the functionality in textual form can be validated by stakeholders or lawyers, and it can be a part of a contract.

Our paper is structured as follows: First, we explain the motivation in terms of the software development process using Normalized Systems and set our goals in Section 2. Then, related work is briefly introduced to provide the necessary context and relevant references to previous work in Section 3. Section 4 describes both a theoretical solution and a technical solution to generate Normalized Systems from textual domain descriptions. The implementation of the text-to-NS pipeline is described in Section 5. In Section 6, the resulting solution is evaluated using a prepared case, and future research is outlined. Finally, Section 7 summarizes the outcomes of our work.

2 Software Development Process with NS and Our Goals

The software development process starts with requirement specifications. Then, based on the methodology (waterfall, iterative, or agile), design, development, and

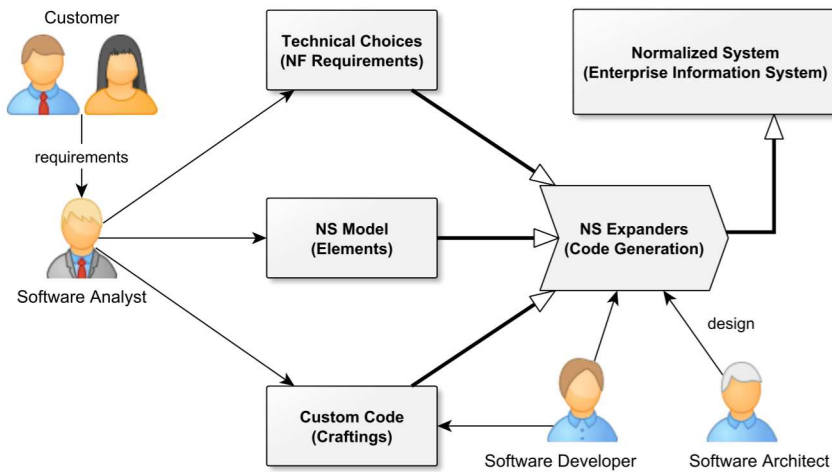


Fig. 1 Software development process with Normalized Systems.

verification occur. Normalized Systems are not bound to any software development methodology; both traditional and agile methodologies can be applied. The process focuses on (re)generation from the specification (models) to evolvable information systems. As shown in Fig. 1, the software analyst creates an NS model based on the domain requirement specification. Then, a prototype information system with generic functionality can be directly generated and verified with the customer. Usually, the requirement specification is refined multiple times during this process. The model is elaborated, and custom features are described. These features are then implemented as code fragments plugged into the generic system. Therefore, even if there are further changes in the model, it can be regenerated while retaining the custom features.

Our research pursues the following goals:

- G1:** Enhance TEMOS to utilize NLP to produce NS models from textual domain descriptions.
- G2:** Add the possibility of exporting NS models from TEMOS into formats used by NS tooling.
- G3:** Streamline the prototyping of NS from text.

3 Related Work

This part briefly overviews relevant topics, technologies, and terminology with respect to the goal of our research. It also provides important references to existing previous work.

3.1 Requirements Engineering with NLP – Classification

The idea of supporting requirements engineering by tools based on the linguistic approach is a topic addressed in papers taking into account at least one of two goals – to extract a model from the textual description and/or to clarify and improve the textual description. In papers [8] and [7], Kof broke down the NLP approach into three groups:

- The first one is related to lexical methods – methods that do not rely on basic NLP approaches like part-of-speech tagging nor any other parsing. These methods perceive the text as a sequence of characters and look for terms (subsequences) that occur repetitively.
- Syntactical methods typify the second group. These methods use part-of-speech tagging and looking for special sentence construction. Based on that, they are able to distinguish objects and relationships.
- The last group represents semantic methods – methods that interpret each sentence as a logical formula or a partial model. The goal is then to look for predefined patterns or structures.

Moreover, the semantic methods should be supported by predefined patterns/structures created by humans or via machine learning [27].

3.2 Requirements Engineering with NLP – Related Work

In the paper by Rolland and Proix [18], the authors introduced a tool called OISCI. This tool processes the French natural language. The approach presented in the paper targets the creation of the characterization of the parts of the sentence patterns that will be thereafter matched. OISCI also uses a text generation technique from the conceptual specification to natural language for validation purposes.

Linguistic assistant for Domain Analysis (LIDA) is a tool presented by Overmyer, Lavoie, and Rambow [16]. *LIDA* is conceived as a supportive tool – it can recognize multi-word phrases, retrieve the base form of words (stemming and lemmatization), present the frequency of words, etc. – but it does not contain algorithms for automatic recognition of model elements. The decisions about modelling are fully user-side, i.e., the user marks candidates for entities, attributes, and relations (inclusive operations and roles).

TextReq tool [1] is implemented using *The Natural Language Toolkit (NLTK)*. This paper uses parsed trees of sentences, including part-of-speech tagging and dependency recognition, to recognize entities and attributes. From the papers mentioned above, the concept presented in this paper is the closest one to our approach.

Visual Narrator tool [17] focuses on textual user stories and generates conceptual models as OWL ontologies. It is based on the *spaCy* NLP framework, similar to our implementation. It requires user stories written in the form with indicators (e.g., *I want, I can*). These are parsed in the sense of part-of-speech tagging and dependency recognitions for further processing.

The same approach of using part-of-speech tagging and dependency recognition is presented in the paper by Rooijen et al. [28], too. There, the tool is called *REACT* (Requirements Extraction and Classification Tool).

A more detailed overview of existing tools is presented in the recent paper by Zhao et al. [30].

3.3 TEMOS

As the extension of the TEMOS tool is set out in goals **G1** and **G2** of our work, we present this tool in a separate section.

TEMOS supports requirements engineering concerning both goals – model extraction and textual description clarification. As we describe in the paper [20], we derive the UML model using the grammatical inspection of the textual form of requirements. The process consists of three phases. During the first phase, the text is parsed, and an internal model is created. During the second phase, we search for patterns indicating ambiguity [21], incompleteness [22, 23], and inconsistency [29]. We also confront the problem of domain facts and rules that domain experts find so obvious that they fail to mention them. We denoted it as a problem of default consistency rules [24].

TEMOS focuses on making textual requirements specifications more precise and less inconsistent. To achieve it, we use the semantic information derived from the complete textual description of requirements specifications with the help of the simultaneously constructed internal model. Concerning the categorization provided in Section 3.1, our approach combines syntactical and semantic methods.

The tool TEMOS was primarily created for UML class model generation. Anyway, thanks to the internal model (created incrementally by processing sentence by sentence), we are able to convert semantics from the internal model to different target forms. In the past, we also generated also SHACL shapes [19].

3.4 Normalized Systems

Normalized Systems (NS) theory [11] describes how to design and build complex systems that can easily adapt to new changes using fine-grained modular structures and the elimination of combinatorial effects. Although the theory is applicable in various domains, and some works focus on that (e.g., [15] or [26]), it is the most visibly used in software engineering. The theory itself describes how to build software systems without combinatorial effects. It does so by applying four basic principles: Separation of Concerns, Action Version Transparency, Data Version Transparency, and Separation of States. The modular structure consists of so-called NS Elements of five kinds: Data, Task, Flow, Trigger, and Connector. The specification of an NS application consists of technical details and Elements models encapsulated as Components. Each component can depend on other components and describes a specific part or concern of the system. [3, 13]

With an NS application, specifications consisting of components and elements, NS Expanders can be used. During the expansion, selected code templates are filled with information from the specification and previously harvested fragments (custom code or craftings). The generated source code can then be used to build and run the application as well as for further customizations. As such craftings are always harvested, both the specification and code templates may change, and the application can be regenerated. The craftings may bring combinatorial effects

and other issues, but it has been empirically shown that there is a maximum of 10% of craftings in a codebase. [4,6]

The NS metamodel describes the elements, components, applications, other entities, their value fields and link fields. An important aspect of the metamodel is its meta-circularity, i.e., the NS metamodel is itself an NS model [10]. For example, a Data Element that describes structural entities is in the metamodel itself a Data Element. It has significant implications with respect to the NS tooling that supports the modelling, composing specifications, expanding, and managing NS applications. Those tools can be expanded as NS applications themselves which ensure their evolvability and maintainability.

3.5 Transformations into Normalized Systems

According to the NS metamodel, the goal of this work is to transform a natural text into an NS elements model. The previous research on transformations or mappings between NS models and some other knowledge representations can provide valuable insights. The bi-directional transformation between NS models and OWL ontologies [25] defines the mapping between the NS metamodel and core constructs of the Web Ontology Language (OWL). It also describes how a tool is built to execute transformations according to the mapping in both directions. Instead of finding natural language patterns in text descriptions, it deals with knowledge graphs and transforms from OWL to NS by looking up specific nodes and edges mapped to the NS Elements.

The NS Elements are tightly related to the concept of projections, as also demonstrated in [25] that builds a way of a new type of projections. The XML representation that is commonly used for NS models serialization is a specific projection. Thanks to the NS metamodel's meta-circularity, the projection allows to import and export of the NS metamodel (and any of its instances) using XML [13]. Our work will focus on working with NS models in their XML format.

3.6 Summary

In the context of the presented related work, we follow the idea of the TEMOS tool regarding text processing. The objective is similar to other presented tools: extract information from the perspective of requirements engineering. The tools/works differ in the used NLP framework and the used features of the annotated text. We reuse the method of grammatical inspection that we will describe later.

The standard output of the TEMOS tool is the UML diagram. In this work, where we extend this tool, we try to convert our internal model into a new form – an NS elements model.

Following the existing works regarding Normalized Systems generation, we connect the research of model generation from the text and the Normalized Systems generation from the model.

4 Our Approach in Generating NS Elements from Textual Domain Description

In this section, we describe how we use natural language processing methods to convert textual information into our internal model. This model is in the next phase transformed into NS metamodel.

4.1 The Method of Grammatical Inspection

Let's consider the whole input text as a collection of sentences. A sentence is a couple $S = (T, D)$ where T is an ordered set of tokens t_i and D is a set of dependencies. This structure is created as follows.

We use a quite typical pipeline of the text mining process that includes these phases:

<i>tokenization</i>	identifies tokens t_i where a token is a word or interpuncton,
<i>sentence segmentation</i>	composes the sentence S based on the tokens representing interpuncton,
<i>part-of-speech tagging</i>	resolves the part-of-speech tag of each token t_i ,
<i>lemmatization</i>	identifies the word's lemma (sometimes also called as dictionary form),
<i>dependency recognition</i>	constructs set D , where the dependencies, i.e., relations between tokens, are mapped,
<i>co-reference recognition</i>	links tokens represented by a pronoun to their original referent (the same person or thing typically represented by a (proper) noun).

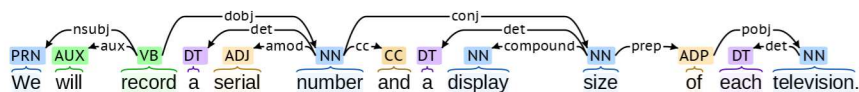


Fig. 2 Example of a sentence with attribute candidates.

A result of this pipeline is the annotated original text of requirements that we use as a source for our algorithm. An example of the annotated sentence is in Fig. 2. For our purpose, each token t_i has the following properties: original text presented in a sentence, part-of-speech tag, lemma, and optional co-reference link to another token. A dependency d_j is a relation between two tokens t_1 and t_2 having a type such as subject, object, etc. In our examples, we use types provided by spaCy trained models for English¹. It is a subset of Universal Dependency Relations for English².

We use the method of grammatical inspection improved by our sentence patterns definitions to process the annotated text. On behalf of this method, we define sentence patterns. These patterns use the grammatical structure (primarily part-of-speech tags and dependencies), and we describe them in the next sections.

¹ <https://spacy.io/models/en>

² <https://universaldependencies.org/u/dep/index.html>

4.2 Suitable Patterns

The idea of patterns is to use the grammatical role of words (mapped to tokens), i.e., subject, object, etc., to indicate parts of the model to be created. For example, both the subject and the object are candidates for an entity or an attribute in our model.

The whole text is checked against our collection of patterns – sentence by sentence. The extracted text fragments matching one of our patterns are processed by our internal preliminary model manager. The model manager checks the text fragments and transforms them into model parts if they satisfy all conditions. Some of them are listed in the following section.

We reuse and extend the collection of patterns presented in papers listed in Section 3.3. However, before we present the concrete patterns, we will first show the structure of our model to illustrate how we map the extracted parts of the text.

4.3 Internal Preliminary Model (Manager)

In the end, the information that we would like to extract from the requirements is a collection of entities (classes), their attributes, and relations between the entities. Thus, the goal is to transform the *syntactic analysis* performed on a text of the requirements into a *semantic form* – a model.

The output of our text mining process is an internal preliminary model represented by a couple $M = (E, R)$ where E is a set of elements and R is a set of relations. Each element e_i has the following properties:

<i>root lemma</i>	the lemma form of the main word (token) representing element, e.g., <i>room</i> if we have element <i>hotel room</i> ,
<i>full lemma</i>	the lemma form of the whole element, e.g., <i>hotel room</i> ,
<i>is entity candidate</i>	flag indicating whether the element should be modeled as an entity or as an attribute,
<i>is unique</i>	a flag indicating whether the requirements state the element as unique or not.

Each binary relation $r_j \in R$ links two elements $e_k, e_l \in E$ (where $k \neq l$) or one element e_k recursively, and it is represented by the label and cardinality type. For our purposes, we distinguish the basic cardinality type (single or multiple) of each actor of relation.

The goal of the internal model (manager) is to store the extracted information and to provide constraints such as:

- text fragments representing the same semantic part of the model (e.g., *hotel room*) are stored as only one element e ,
- excluded text fragments listed in the configurable list are not processed,
- elements marked as entity candidates (i.e., because we already found this semantic indicator in the previous text part) can not be changed to attribute candidates; the opposite change from attribute candidate to entity candidate is fully legit because we can find that original attribute candidate has custom attributes or other entity symptoms in the different parts of the text.

The list of excluded text fragments excludes such parts from storing them in the model. The typical reason is that the meaning of these text fragments is too general. For example, we do not want some introductory sentence like “*This document describes requirements of...*” to generate entities *document* and *requirement*, and connect them with the relation *describe*. On the other hand, in the requirements describing a document management system, the word *document* will be one of the key entities. Because such an exclusion is highly context dependent, we leave the list of excluded text fragments configurable. The default list has been created manually by our previous experience with requirements processing.

4.4 Suitable Patterns – Parts of Internal Model Recognition

Now, we know the target structure that is the output of the grammatical inspection method, so we can present the specific patterns that we use to create and populate the internal model.

We divide this section into sub-sections representing specific text structures recognition that we find interesting.

4.5 Suitable Patterns – Triple Recognition

The basic semantic information of sentences is hidden in triples. In our case, a triple consists of a subject, a predicate, and an object similar to the Resource Description Framework (RDF). In Fig. 3, there is an example structure of annotated text segment that corresponds to a triple recognition pattern. The predicate is represented by a verb (part-of-speech tag property of a token). Next, there are two following dependencies. The first one is called *nsubj*, and it represents the nominal subject in our triple. The second one is called *dobj*, and it represents the dependency object in our triple.

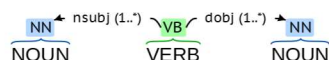


Fig. 3 Basic triple pattern.

When such a pattern is matched, we can identify:

- entity candidate element e_1 represented by the nominal subject part of a sentence,
- entity candidate element e_2 represented by the dependency object part of a sentence, and
- relation r where the label of the relation is represented by the lemma of the verb token.

We can notice that the dependency type label consists of $1..*$ part. It represents cardinality. We require at least one subject and one object. In any case, there can be more subjects or objects. In such cases, we generate more triples with the same predicate.

The second level of cardinality is directly represented by the concrete subject or object. We need to distinguish between the sentences “*the business group owns a hotel*” and “*the business group owns hotels*.” This information is a part of the part-of-speech tag where **NN** represents a singular noun and **NNS** represents a plural noun. Anyway, this is not enough, and we need to consider other linguistic expressions of plurality, e.g., “*the business group could own more than one hotel*” or “*the business group owns at least one hotel*”. As shown in Fig. 4, this can be recognized via **nummod** (numeric modifier) dependency and other modification dependencies representing the meaning of *minimum/maximum, at least, more than/less than*, etc.

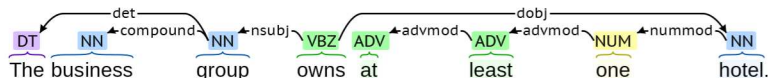


Fig. 4 Example of the cardinality recognition.

4.6 Suitable Patterns – Triple Recognition – Challenges

The pattern presented in Fig. 3 reflects the fundamental sentence fragment – the bare semantic triple. However, the situation in textual requirements specification is far from being that simple. We need to face the challenges originating from passive voice, modifier in the form of prepositions, indirect subject, negation, and many others.

In Fig. 5, there is an example of the passive voice sentence. We are now curious about **nsubjpass** dependency instead of **nsubj** dependency used in the active voice. We can note that we also miss the direct **dobj** dependency. This time, there is **pobj** dependency that presents an object of a preposition. Also, our object (the word *guest*) is not directly connected to the predicate (the word *place* (the lemma form)), so we need to check another dependency called **agent**.

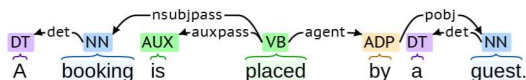


Fig. 5 Example of the passive voice.

When we match this pattern, we consider the information as a triple transformed to active voice. In Fig. 5, it is the triple $\langle \textit{guest}, \textit{place}, \textit{booking} \rangle$.

A similar situation occurs when the **agent** dependency is replaced by **prep** dependency. The example is shown in Fig. 6. This time, we expect the triple $\langle \textit{room}, \textit{relates (to)}, \textit{booking} \rangle$.

Similar to the basic triple pattern in Fig. 3, we can identify two entity candidates and one relation.

To make it easier, in the examples of triples recognition, we present the root token (word) representing entity or attribute candidates. However, as defined in our internal model structure in Section 4.3, we recognize also the full lemma.

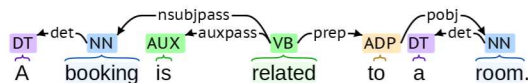


Fig. 6 Example of the passive voice.

Often, the entity or attribute name is a composition of multiple nouns or a noun and an adjective or both. This situation can be recognized by patterns checking the root token relations (dependencies): **compound** and **amod** (adjectival modifier). Both representatives are present in Fig. 2: *serial number* (adjective + noun) and *display size* (compound nouns).

To illustrate a topic of negation in sentences, we reuse methods presented in [29]. The negation can be represented by a standard negation of the verb (e.g., *A user can't...*, *A system is **not** working...*) or another negation modifiers (e.g., ***No** user can...*, *A user can **never**...*).

When presenting challenges, we also need to say that not every recognized triple necessarily brings new semantic information. Some of them can consist of the excluded word(s) that we won't track; other of them can repeat already stated semantic information in the model. That is the responsibility of the model to eliminate possible duplicates.

4.7 Suitable Patterns – Attributes Recognition

In this category, we present two typical representatives. In Fig. 7, there is an attribute(s) mapping via relation with the verb *have*. In that case, the root triple of the sentence is $\langle user, has, username \rangle$. The subject (*user*) and the object (*username*) are mapped to elements in our internal model. The word *user* is mapped as an entity candidate and the word *username* is mapped as an attribute candidate.

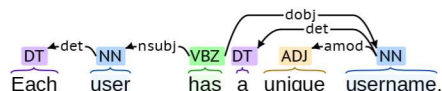


Fig. 7 Example of a sentence with attribute candidate and the verb *have*.

We can see that the sentence structure also conforms to the previously shown pattern regarding the general triple. So, we need to check the lemma of the verb first to prioritize the attribute pattern over the general triple pattern.

In Fig. 2, there is the second attribute-mapping pattern representative. This time, the attribute indication hint comes from the combination of the verb *record* and the preposition *of*. The object of the preposition (*television*) is an entity candidate, and the objects of the triple are attribute candidates. This pattern is variable per verb. In addition to the verb *record*, it can also be a verb *register* or the verb *write down*.

As mentioned in the triple recognition challenges, we need to take into account also semantically indirect subjects. Such a case is shown in Fig. 8. In this example sentence, the syntactic subject represented by the pronoun *we* is not semantic

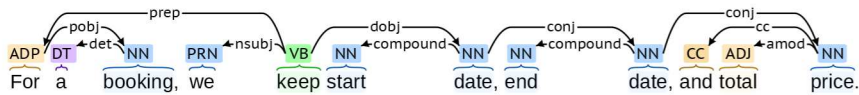


Fig. 8 Example of a sentence with a semantically indirect subject representing an entity.

rich in our case. We would like to extract triples in the form $\langle \textit{booking}, \textit{keep}, \textit{start date} \rangle$ and variants for other attributes. This sentence structure can be matched by a pattern detecting **prep** (preposition) dependency and following **pobj** (object of the preposition) dependency to the expected entity candidate (a *booking* in our case). Furthermore, the extracted relation *keep* is via mapping transformed into the relation *have* that indicates that *start date* is an attribute candidate.

The example sentence in Fig. 7 also hints us how we can determine the *is unique* property of element *e*. We can use a pattern that checks the presence of the adjective *unique* and its connection to our entity or attribute candidate.

4.8 Suitable Patterns – Hierarchy Recognition

In the UML class diagram, we can model the inheritance relationship. On the level of the NS metamodel, we express inheritance in the form of standard relation. We will discuss it in Section 4.10 in the context of mapping our internal model to NS metamodel. In any case, we benefit from the recognition of hierarchy in the text.

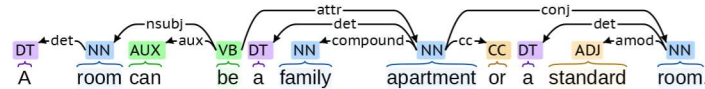


Fig. 9 Example of a sentence representing hierarchy of entities.

In Fig. 9, there is an example sentence representing a hierarchy of entities. The corresponding pattern checks the **nsubj** (nominal subject) dependency, the main verb have to be a form of the verb *be*, and there have to be the **attr** (attribute) dependency instead of the **dobj** (dependency object) dependency. All the subject and object(s) are mapped as entity candidate elements. The subject represents the base entity, and the object(s) represent(s) the sub-entity(ies). We map this hierarchy in our model and convert it to the specific relations based on the target output model as mentioned. There should be more than one sub-entity. We can find the other candidates via **conj** (conjunct) dependency.

In the words of triples, we can identify two triples here: $\langle \textit{family apartment}, \textit{is-child-of}, \textit{room} \rangle$ and $\langle \textit{standard room}, \textit{is-child-of}, \textit{room} \rangle$.

4.9 Internal Model Check

It is possible to utilize our checks for incompleteness and inconsistency detection, as proposed in our papers [22], [23], and [29]. It is beneficial to run these checks

before we start creating the model. After the checks that produce warnings, the analyst can refine the textual input requirements.

We reuse the method of synonym concepts recognitions on elements to reduce ambiguity [21]. When for an element e_i there is identified a synonym element e_j , a warning is generated. For example, with a set $E = \{\text{booking, guest, reservation, ...}\}$, a single warning reports that booking and reservation are synonyms and these concepts might be merged. This should be recognized, e.g., by on-line semantic network *ConceptNet*³. This network provides a list of synonyms for the term being queried.

4.10 Mapping to NS Metamodel

When the input text is processed, we have our internal model M ready for a conversion to the NS model. The conversion has the following steps.

1. For each element e_i from the set of elements E from our model M :
 - if the *is entity candidate* property equals true, we convert e_i to *data element* in the created NS model,
 - otherwise, e_i is attribute candidate, and we assign it as *value field* to the corresponding *data element* representing entity.
2. The type of data element is *primary* by default. However, if the root lemma of e_i is *type*, then the data element type is *taxonomy*.
3. We convert each binary relation r_j from the set of relations R from our model M to *link field* of data element created from e_k (in the first step). We set *link field* properties as follows:
 - *link field type* based on the cardinality type of r_j ,
 - *required* also based on the cardinality type of r_j ,
 - *target* to the already created data element e_l (in the first step).
4. We convert each recognized hierarchy relation. According to the NS theory, inheritance causes combinatorial effects, i.e., it is considered as an obstacle in evolvability. Therefore, inheritance must be modeled using link fields. Such a link field is created on the data element representing sub-entity e_k of parent entity e_l . The target of a link field is the data element representing e_l . This time, the required property is always true, and the cardinality is singular.
5. The last step is the enhancements phase. We describe it in the next section.

4.11 Enhancements Phase

After composing a model of data elements, we add several steps to apply conventions from NS modelling and enhance the resulting model. The first step is to handle the relations between primary and its corresponding taxonomy data elements. It is a pervasive pattern to have a taxonomy data element with name suffix *Type* with just a single attribute *name*. In other modelling languages such as UML, it would become an enumeration. However, in terms of the NS theory, enumerations are blocking the evolvability (e.g., when we need to add fields to enumerated items). We noticed that the type is specified just as a “type” without

³ <https://www.conceptnet.io>

any additional information in the textual requirements. When such a value field is created, it is changed in the enhancements phase to a link field linking the newly created taxonomy data element with name suffix *Type* and a *name* value field. For example, if there is a *Vehicle* data element with *type* value field, it is changed to a link field pointing to new *VehicleType* that has a *name*.

The second step is again related to the taxonomy data elements. We identified that some texts describe both primary and taxonomy data elements but not the relation between them. It is fixed by simply looking for such data elements with missing relation to the corresponding taxonomy data element (if it exists). For each of them, a new link field in the primary data element is created. The final step of enhancements adjusts link field names using the conventions of NS modelling. According to the mapping, the names of the field are taken from triples that do not allow to match bi-directional links in NS models. If there is just a single link field to the target data element, it is named by the target element's name. In the case of the many-side of the relationship, i.e., when it contains a collection of target element instances, a suffix is added to express plural. For example, the value field *drives* from *Person drives Vehicles* is renamed to *vehicles* but we keep *drives* as a description of the field.

4.12 Exporting NS Elements

The classes for representing data elements, their value and link fields, and other properties are expanded directly from the NS metamodel, which is meta-circular. The XML format is required to allow the transition of NS models in TEMOS to NS models for expanders. Due to NS tooling's versatility, we were able to implement expanders for the data-classes in Python and the XML serialization. There are two benefits when using the approach with expanders in this case. First, whenever the NS metamodel is updated, it is possible to re-expand the classes and related serialization for TEMOS. Second, although we currently focus on the structural part of the NS metamodel (i.e., data elements and the constructs around them), expansion works with full NS metamodel. It will simplify future development when we can focus, for example, on task and flow elements. TEMOS has been extended with a command-line interface that takes a text as input and writes a set of XML files with data elements according to the NS tooling expectations.

The set of XML files generated from TEMOS can be added to an existing NS component or to a new prepared one. It is not possible to generate a component fully from functional requirements. It contains various implementation and environment details, e.g., which source base is used or the version or qualified name of the component. These details need to be provided in other NS tools, or an example component XML file can be adjusted accordingly. Such a component can then be imported for further refining, e.g., in NS Modeler, or directly used for expansion.

We used the sentences introduced in Fig. 5, Fig. 6, Fig. 9, and Fig. 8 as requirements to generate an NS model of data elements. Fig. 10 shows the resulting model with five data elements. There were no fields for *Guest* and *Room* mentioned in the text. Both *Standard Room* and *Family Apartment* have the relation to parent element *Room* as they form a hierarchy. Finally, there are several fields (value and link) for the *Booking* data element.

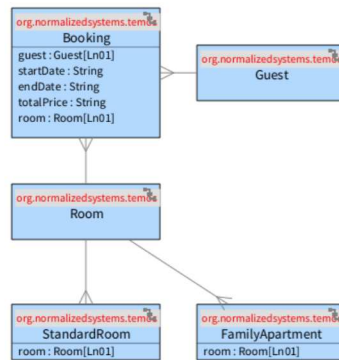


Fig. 10 Generated data elements for a hotel example.

Fig. 11 Booking form for expanded NS application.

4.13 Generating Information System Prototypes

The expansion is possible directly with an NS component filled by data elements from our transformation. The expanded information system is possible to build and deploy just by executing a set of commands or by clicking the button in the NS tool called Prime Radiant [13]. The resulting system is a basic CRUD application but fully operational and can serve as a prototype for further requirements elaboration. Fig. 11 shows a form based on the *Booking* data element. We manually added value fields *name* for *Room* and *Guest* as data elements without any attribute are meaningless.

With the prototype, it is possible to change the NS model directly in NS Modeler or Prime Radiant and quickly re-expand, build and re-deploy the application. If the requirements are more specific, even custom code fragments called craftings can be added to the expanded code base. NS comes with a principle of harvesting those craftings, so they are not lost upon re-expansion. Whenever there is a change of the requirements text, it is possible to regenerate XML files of data elements, but if other changes were made, those would be overwritten. On the other hand, if there are some harvested craftings or other than data elements in the component, such additional elements stay intact.

5 Text-to-NS Pipeline Implementation

The text processing part is an extension of the TEMOS tool, and it is powered by the *spaCy*⁴ NLP framework, version 3.2. We use a pretrained model called *en_core_web_trf* in version 3.2.0 (available together with the spaCy installation) to process text written in English. The text processing logic is written in Python. Both the spaCy version and the model version are the latest versions available at the time of writing this paper.

To streamline the process of information system prototyping from textual functional requirements (set as goal **G3**), we prepared a script for automating it. As input, it takes the textual requirements together with a directory in which the target NS component is located and the Prime Radiant is located (for expansion and deployment). Then, it executes the following steps:

1. Run TEMOS to create data elements as XML files in the component.
2. Expand the information system using Normalized Systems.
3. Build the information system from an expanded code base.
4. Deploy/run the information system.
5. (optional) Open a web browser with the running system.

Steps 2 and 3 are performed by executing Maven targets, as the Normalized Systems use the Java programming language. For deploying and running in step 4, the system is managed through Prime Radiant, which supports multiple deployed bases and various options (e.g., database management). As stated, the system can be easily reified and recreated after stopping.

6 Evaluation and Discussion

In this section, we evaluate our approach using textual requirements from three different sources: first, the EUrent case provided by NSX, second, a set of requirements from the natural language requirements processing (NLRP) benchmark provided by Karlsruhe Institute of Technology, and third, a set of requirements from the Public REquirements (PURE) dataset.

We describe each input data text with the number of sentences, the number of words, and two indices of readability – the *Automated Readability Index* (ARI) and the *Gunning Fog Index* (GFI) – that we obtained using the *Free Text Complexity Analyzer* online tool⁵. In the evaluation, we analyse precision and recall considering relevant (expected) model constructs and retrieved (generated) model constructs. In our case, they are defined as:

$$precision = \frac{|\{\text{relevant model constructs}\} \cap \{\text{retrieved model constructs}\}|}{|\{\text{retrieved model constructs}\}|}$$

and

$$recall = \frac{|\{\text{relevant model constructs}\} \cap \{\text{retrieved model constructs}\}|}{|\{\text{relevant model constructs}\}|}$$

⁴ <https://spacy.io>

⁵ <https://www.lumoslearning.com/llwp/free-text-complexity-analysis.html>

A model construct in this analysis includes data elements (both primary and taxonomy elements), link fields, and value fields.

We discuss the challenging situations and issues regarding missing or incorrectly generated model constructs. Furthermore, we discuss general software development aspects, revisit our initial goals, and propose future work.

6.1 Evaluating with the EUrent Case

To assess our approach, we use an official example called the EUrent Case [12]. NSX has used it for several years as a training example for new analysts. In 45 sentences (522 words, ARI 4.07, GFI 7.48), 40 data elements are described, of which two are of the history type (but are not covered in the textual description) and nine are of the taxonomy type. This example also contains different kinds of value and link fields. We used our pipeline on the textual description of the EUrent case to generate an NS model and expand the corresponding information system prototype. This section discusses the differences between our generated NS model (presented in Fig. 12) and the reference model created by analysts who have experience with Normalized Systems. In NS visualization, taxonomy data elements use a red background, whereas primary data elements are blue.

6.1.1 Missing Data Elements

We need to take into account that the NLP framework (based on spaCy) works with pretrained models, and the resulting annotations should differ when a specific sentence is used alone or in the context of a paragraph.

This case occurred in our evaluation, where a specific dependency had a different root token than we expected.

6.1.2 Missing Value Fields

The attribute *average price* is missing because it was not matched by any of our attribute patterns.

6.1.3 Mistaken Value Fields

The reason for the incompletely identified value field *damaged* is the unmatched condition sentence connected to the word *flag*.

6.1.4 Missing Link Fields

NLP recognition fails on passive-voice sentences with the verb *relate*. Similar to the first point, there is a difference when the sentence is used alone or in a paragraph. In the case of a paragraph, the word *related* was incorrectly identified as an adjective instead of a verb. This issue concerns almost all missing link fields.

6.1.5 Excess Elements

Fortunately, we did not find any excess elements. As mentioned in Section 4.3, a configurable list of excluded text fragments can be helpful here. There is a possibility of using human intervention to exclude words or text fragments from processing.

In our case, the requirements contain the word *today*. Without excluding this word, it should be identified as an entity candidate and transformed into a data element in the NS model. We show this situation in Fig. 12 with a dashed rectangle.

6.1.6 Taxonomy Data Elements

Due to the enhancement phase, all of the taxonomy data elements were correctly identified and completely matched. The missing information was deduced despite

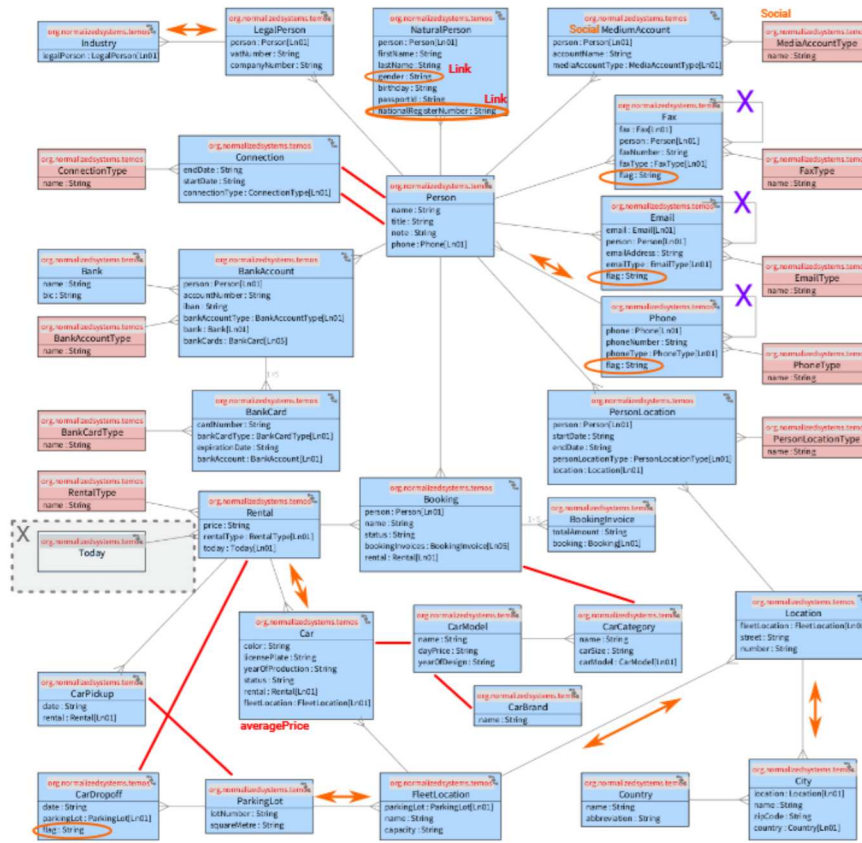


Fig. 12 EUrent case generated diagram with evaluation.

the ambiguity and incompleteness in the text, e.g., not mentioning the name attribute or stating that there is only a type for other entities. Moreover, no extra taxonomy data elements were generated. Therefore, for these nine data elements, our method achieved the best possible result.

6.1.7 Degradation of Links to Values

When compared to the model created by an analyst, we differ in the fields of *Natural Person*. In our case, both fields – *gender* and *national register number* – are value fields because they conform to our attribute patterns. However, experienced analysts model them as data elements and link fields from *Natural Person*. For example, an analyst may use a *name* of *Gender* that is not stated in the requirements, so this is an example of incompleteness.

6.1.8 Differences in Naming

Our patterns were created for sentence structures in which the adjective is connected to the root token (word) of an attribute or entity candidate. Nevertheless, thanks to this case study, we found that the adjective can also be connected to the first token (word), e.g., *social media account* (because *social media* is a well-established phrase) vs. *new printer cartridge*.

6.1.9 Evaluation Summary

As described in the paragraphs above, there are differences between the original and the generated NS model. Some of these differences are caused by ambiguity, incompleteness, or inconsistency in the textual requirements. Moreover, the software analysts – based on their experience – may add constructs that are not present in the text. Table 1 summarizes the numbers of constructs that are matched, missing (not present in the generated model), excessive (not present in the original model), and mistaken (present but with different properties). We show the differences in the diagram of the generated model in Fig. 12: the red lines represent missing links, the orange lines with arrows indicate a reversed direction, the purple crosses denote construct excess, and the orange ellipses mark mistaken fields. To compute the precision and recall metrics, the number of missing constructs represents false negatives, excessive and mistaken constructs are considered false positives, and matches are true positives. Then, the retrieved elements are the union of the matching, excessive, and mistaken constructs. Finally, the relevant elements are matched with missing data.

The method we propose achieved a precision of approximately 87% and a recall of approximately 92%. The overall score (the ratio of matching model constructs to all expected model constructs) is 81% for this case. The result is usable as the first prototype for further refinements in other NS tools as intended. However, it counts incompleteness and ambiguity as having a negative impact on the method. If we ignore the differences caused by such issues with the input text, we obtain an overall score of 85%. When we ignore the differences that are caused by analyst experience (e.g., taking *Gender* as an element instead of a value field), it further increases the score to 89%.

6.2 Evaluation with NLRP Benchmark Texts

By applying the same evaluation technique as in the EUrent case, we also analysed the results for selected texts from the NLRP Benchmark site⁶. We selected seven requirement descriptions relevant to software information systems that described the structural aspect of the domain in the form of natural English text. We did not use descriptions that were written as simple bulleted lists of functional and nonfunctional requirements or descriptions focused on processes. Moreover, the examples had various length and complexity metrics. The dataset does not contain any reference models; however, there are written training data (primarily for students), so we were able to prepare reference models ourselves.

A summary of the results is shown in Table 2. We verified that the main issues are related to the semantic content of the text rather than the complexity of the sentences. For example, the Movie Theatre case has a low score because it is described from a user navigation perspective in some systems. The better results (the Address Book, Elevator, and EUrent cases) are pure descriptions of the problem domain – the entities, their properties and relations. When a customer is guided to describe a case in such a manner, precision and recall can be expected to be approximately 80%.

6.3 Evaluation with the PUBLIC REquirements (PURE) Dataset

The third dataset evaluated is a dataset of public requirement documents called PURE [5]. We select a subcollection of documents that consist of functional requirements concerning domain descriptions. In this evaluation, we consider the precision metric only. The dataset does not contain any reference models. Moreover, some of the requirements are already quite complex, and we believe that different analysts can model them in different ways based on their experience and expertise. Therefore, it is difficult to evaluate recall objectively.

A summary of the results is given in Table 3. We follow this with a discussion of our findings for this last evaluated dataset. In the evaluation, we also address documents containing functional specification sections with more than 5,000 words (the largest document, the Inventory 2.0 case, has 8,116 words).

Overall, we confirmed the idea of the usefulness of the list of excluded words and text fragments. We expanded this list with new elements. One of them is the

⁶ <http://nlrp.ipd.kit.edu/index.php/Category:Language:English>

Table 1 EUrent case evaluation.

	Match	Missing	Excess	Mistaken	Precision	Recall
Primary DEs ¹	26	2	0	1	96.30%	92.86%
Taxonomy DEs ¹	8	0	0	1	88.89%	100.00%
Value Fields	58	1	0	6	90.63%	98.31%
Link Fields	26	7	3	6	74.29%	78.79%
Total	118	10	3	14	87.41%	92.19%

¹ DEs = Data Elements

Table 2 Evaluation of additional cases from the NLRP Benchmark.

Case	Words	Sentences	ARI ¹	GFI ²	Precision	Recall
Address Book	417	15	12.9	13.6	81.82%	78.26%
Elevator	180	10	10.4	12.3	81.25%	100.00%
Hotel	541	24	3.7	7.7	78.57%	91.67%
Movie Theatre	176	9	8.1	11.0	86.21%	78.13%
Library	216	15	5.5	10.2	85.71%	66.67%
Ships	49	5	6.8	11.3	100.00%	75.00%
Trains	78	6	5.7	11.9	88.89%	84.21%
Average					86.06%	81.99%

¹ ARI = Automated Readability Index; ² GFI = Gunning Fog Index

word *information*. For example, in the sentence “It shows information such as cases assigned, . . .”, we want to skip the word information and map the case to the entity referenced by the pronoun *it*. Regarding references, some cross-references targeting section headers or words in more than 2 previous sentences are also challenging to process.

Another point concerning the list of excluded words is the problem domain itself. For example, the word *value* is typically processed to enrich the model with restrictions regarding the data type that represents the value, e.g., cardinality. In this case, the word *value* is not mapped to any data element. However, in a specification describing sensors, *value* should represent a standalone data element in the sense of the recorded value of a sensor. This recorded value should have custom attributes such as a timestamp and some numeric values that a sensor measures.

We also need to take into account text formatting that decreases recognition success. When processing plain text converted from document formats with rich formatting, we lose information about sections (headers), lists, and tables. The tables are the most challenging elements. On the other hand, we can quite successfully recognize standalone text fragments representing headers and bullets representing lists.

6.4 Software Development Aspects

Our approach can be seen as an extension of the methods used in model-driven development. Traditionally, a software analyst takes textual requirements to produce conceptual models that can be used as a basis for designing or, in model-driven development, generating a software system. Even if consistency between such models and systems is assured, this does not help in maintaining the models with respect to textual requirements that can also change over time. This results in manual work for the analyst. Moreover, as each analyst may use different approaches and have different experience, they may model the same thing differently.

With the transformation that we proposed and implemented; a textual description is always transformed into the same model. When the text of the requirements is changed, the transformation can be executed again, a new NS model can be

Table 3 Evaluation of additional cases from the PURE dataset.

Case	Words	Sentences	ARI ¹	GFI ²	Precision
0000_cctns	394	21	14.49	16.13	74.07%
1999_multi-mahjong	1759	88	11.14	10.56	69.70%
1999_tcs	5855	296	13.95	14.40	71.88%
2000_nasa-x68	5455	358	13.96	12.44	68.42%
2001_elsfork	1704	92	13.06	14.05	61.84%
2001_npac	2453	115	14.22	14.56	65.31%
2002_evla-back	844	44	12.58	14.42	58.33%
2002_evla-corr	955	41	15.84	17.05	69.57%
2002_sce-api	2108	76	16.82	15.67	79.52%
2003_pnnl	931	45	15.73	16.70	63.64%
2003_tachonet	492	19	16.96	15.77	81.82%
2004_colorcast	1199	66	12.62	13.49	67.69%
2004_e-procurement	1683	90	11.82	12.22	71.19%
2004_grid-bgc	716	56	10.03	10.01	77.50%
2004_ijis	1763	134	13.09	16.79	76.00%
2004_jse	641	21	19.66	15.69	69.23%
2005_grid-3D	196	11	8.72	9.19	66.67%
2005_microcare	3821	133	16.01	15.10	73.57%
2005_nenios	944	82	7.72	10.63	69.57%
2005_phin	2988	110	19.84	10.63	75.76%
2005_pontis	4395	221	13.73	11.48	72.16%
2008_vub	2546	61	26.97	10.20	68.97%
2009_inventory-2.0	8116	851	8.78	9.13	74.10%
2009_library	1974	79	13.42	10.62	71.26%
2010_home-1.3	1135	45	15.04	9.89	67.50%
2010_mashboot	526	26	13.26	13.50	68.18%
Average					70.52%

¹ ARI = Automated Readability Index; ² GFI = Gunning Fog Index

produced, and a new information system can be expanded, built, and deployed. However, it is crucial to keep track of changes made in addition to the data elements coming from the transformation. If one, for example, adds a new value field in NS Modeler or specifies a data child in Prime Radiant, those changes would normally be overwritten. By using a model representation in XML, on the other hand, conflicts can be solved through standard version control systems (VCSs), which are recommended.

Another change that can arise is related to the NS metamodel, despite its stability. As the module used for representing the NS metamodel and serialization is expanded directly from the NS metamodel, it easily adapts to such changes. Nevertheless, if the NS metamodel is changed significantly, the change will also be required to transform from the internal preliminary model. This means that the NS metamodel is changed according to the theory, and the new versions are backwards compatible. On the other hand, some changes can be used to improve

the transformation. For example, suppose a new type of relation between data elements is incorporated (e.g., inheritance or part-whole relations). In this case, new textual patterns can be designed to find such relations in the textual requirements.

6.5 Goals Revisited

At the beginning of this work, we set three goals listed in Section 2. In this section, we describe how we met these goals.

Concerning the first goal, **G1** *Enhance TEMOS to utilize NLP for producing NS models from textual domain descriptions*, we introduced the focus of TEMOS in Section 3.3, and we followed up on the approach of text analysis using sentence patterns. We described the suitable patterns, and we presented examples of them in Sections 4.2–4.8.

Regarding the second goal, **G2** *Add the possibility to export NS models from TEMOS into format used by NS tooling.*, this began in Section 4.10, where we described the approach of mapping the recognized text parts into the NS meta-model. We also showed how to apply conventions from NS modelling during the enhancement phase in Section 4.11. Finally, we described the exporting process in Section 4.12.

The last goal, **G3** *Streamline the prototyping of NS from text*, is described from a practical process implementation perspective in Section 5. We evaluated and discussed the whole prototyping pipeline in Section 6.

6.6 Future Work

The work presented in this paper is ready to be used, and as explained, it is mainly helpful in the initial phase of software development. Nevertheless, there are further potential steps that can extend our approach. From a content viewpoint, we plan to define more patterns for text analysis to produce different NS Elements. For example, subsequent research could focus on descriptions of behavioural aspects in functional requirements. As a result, the NS models generated from TEMOS would also contain task and flow elements.

Concerning usability, the pipeline that we designed can be incorporated directly into the NS tool to avoid the need for executing a script. For instance, the Prime Radiant tool could contain a form for creating or updating a component from textual requirements. The transformation would be executed by simply clicking a button. For a software analyst, it would be convenient to paste text, click a button, and have a running prototype for further discussion with customers, future users, or different stakeholders.

7 Conclusion

This paper presents our solution for generating information systems directly from textual functional requirements using Normalized Systems. It builds on both the existing capabilities of NS technologies and the tool TEMOS [20]. Our approach

of recognizing specific patterns using NLP methods (including grammatical inspection) and incorporating the intermediate transformation layer of an internal preliminary model allows us to produce working information system prototypes. Such prototypes can be used for verifying and clarifying requirements with clients or as a basis for further manual enhancements. We also automated the pipeline from textual functional requirements to a running prototype without any intermediate steps. Finally, the solution also takes evolvability into account, as both the NS metamodel and textual requirements on a software system may change over time.

8 Declarations

8.1 Funding

The research was supported (in terms of funding) by Czech Technical University in Prague grant No. SGS20/209/OHK3/3T/18.

8.2 Conflicts of Interest/Competing Interests

Marek Suchánek collaborates on research with NSX bvba (University of Antwerp spin-off) as being PhD student with topic oriented on Normalized Systems. Herwig Mannert as one of the NS Theory authors is also one of the NSX bvba founders.

8.3 Availability of Data and Material

Not applicable. There are no datasets created in this research. Source codes are available based on a request.

8.4 Code Availability

Source codes are available from David Šenkýř based on a request. The NS tools are available from NSX.

8.5 Authors' Contributions

David Šenkýř designed and implemented the enhancements in TEMOS in terms of NLP and mapping to the NS metamodel.

Realization of the NS module in TEMOS including export functionality and text-to-NS pipeline has been done by Marek Suchánek.

Supervising the work, collaboration on evaluation has been done by Petr Kroha, Herwig Mannaert, and Robert Pergl.

Acknowledgments

The research was performed in collaboration of Czech Technical University in Prague, University of Antwerp, and NSX bvba. The research was supported by Czech Technical University in Prague grant No. SGS20/209/OHK3/3T/18.

References

1. Arellano, A., Zontek-Carney, E., Austin, M.: Frameworks for Natural Language Processing of Textual Requirements. *International Journal on Advances in Systems and Measurements* **8**(3 & 4), 230–240 (2015)
2. Beydeda, S., Book, M., Gruhn, V., et al.: *Model-Driven Software Development*, vol. 15. Springer (2005)
3. De Bruyn, P.: Towards Designing Enterprises for Evolvability Based on Fundamental Engineering Concepts. In: *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, pp. 11–20. Springer (2011). DOI 10.1007/978-3-642-25126-9_3
4. De Bruyn, P., Mannaert, H., Verelst, J., Huysmans, P.: Enabling Normalized Systems in Practice – Exploring a Modeling Approach. *Business & Information Systems Engineering* **60**(1), 55–67 (2018). DOI 10.1007/s12599-017-0510-4
5. Ferrari, A., Spagnolo, G.O., Gnesi, S.: PURE: A Dataset of Public Requirements Documents. In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pp. 502–505 (2017). DOI 10.1109/RE.2017.29
6. Huysmans, P., Verelst, J.: Towards an Engineering-Based Research Approach for Enterprise Architecture: Lessons Learned from Normalized Systems Theory. In: *International Conference on Advanced Information Systems Engineering*, pp. 58–72. Springer (2013). DOI 10.1007/978-3-642-38490-5_5
7. Kof, L.: An Application of Natural Language Processing to Domain Modelling: Two Case Studies. *International Journal on Computer Systems Science Engineering* **20**, 37–52 (2004)
8. Kof, L.: Natural Language Processing: Mature Enough for Requirements Documents Analysis? In: A. Montoyo, R. Muñoz, E. Métais (eds.) *Natural Language Processing and Information Systems*, pp. 91–102. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
9. Laplante, P.A.: *Requirements Engineering for Software and Systems*. CRC Press (2017)
10. Mannaert, H., De Cock, K., Uhnák, P.: On the Realization of Meta-Circular Code Generation: The Case of the Normalized Systems Expanders. In: *ICSEA 2019, The Fourteenth International Conference on Software Engineering Advances*. IARIA (2019)
11. Mannaert, H., Verelst, J., De Bruyn, P.: *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. Koppa, Kermt (Belgium) (2016)
12. NSX bvba: NS course: EU Rent exercises (domain description) (2017, <https://doi.org/10.5281/zenodo.4629503>). DOI {10.5281/zenodo.4629503}. URL <https://zenodo.org/record/4629503#.YFm0Lnv9aiM>
13. NSX bvba: NSX: Normalized Systems (2020). [online]. <https://normalizedsystems.org>
14. Oorts, G., Huysmans, P., De Bruyn, P., Mannaert, H., Verelst, J., Oost, A.: Building Evolvable Software Using Normalized Systems Theory: A Case Study. In: *2014 47th Hawaii International Conference on System Sciences*, pp. 4760–4769. IEEE (2014). DOI 10.1109/HICSS.2014.585
15. Oorts, G., Mannaert, H., Bruyn, P.D., Franquet, I.: On the Evolvable and Traceable Design of (Under)graduate Education Programs. In: D. Aveiro, R. Pergl, D. Gouveia (eds.) *Advances in Enterprise Engineering X - 6th Enterprise Engineering Working Conference, EEWC 2016, Funchal, Madeira Island, Portugal, May 30 - June 3, 2016, Proceedings, Lecture Notes in Business Information Processing*, vol. 252, pp. 86–100. Springer (2016). DOI 10.1007/978-3-319-39567-8_6
16. Overmyer, S.P., Lavoie, B., Rambow, O.: Conceptual Modeling Through Linguistic Analysis Using LIDA. In: *Proceedings of the 23rd International Conference on Software Engineering, ICSE '01*, pp. 401–410. IEEE Computer Society, Washington, DC, USA (2001). URL <http://dl.acm.org/citation.cfm?id=381473.381515>

17. Robeer, M., Lucassen, G., van der Werf, J.M.E.M., Dalpiaz, F., Brinkkemper, S.: Automated Extraction of Conceptual Models from User Stories via NLP. In: 2016 IEEE 24th International Requirements Engineering Conference (RE), pp. 196–205 (2016). DOI 10.1109/RE.2016.40
18. Rolland, C., Proix, C.: A Natural Language Approach for Requirements Engineering. In: P. Loucopoulos (ed.) *Advanced Information Systems Engineering*, pp. 257–277. Springer, Berlin, Heidelberg (1992)
19. Šenkýř, D.: SHACL Shapes Generation from Textual Documents. In: R. Pergl, E. Babkin, R. Lock, P. Malychenkov, V. Merunka (eds.) *Enterprise and Organizational Modeling and Simulation*, pp. 121–130. Springer International Publishing, Cham (2019)
20. Šenkýř, D., Kroha, P.: Patterns in Textual Requirements Specification. In: *Proceedings of the 13th International Conference on Software Technologies*, pp. 197–204. SCITEPRESS – Science and Technology Publications, Porto, Portugal (2018). DOI 10.5220/0006827301970204. URL <http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006827301970204>
21. Šenkýř, D., Kroha, P.: Patterns of Ambiguity in Textual Requirements Specification. In: Á. Rocha, H. Adeli, L.P. Reis, S. Costanzo (eds.) *New Knowledge in Information Systems and Technologies*, vol. 1, pp. 886–895. Springer International Publishing, Cham (2019)
22. Šenkýř, D., Kroha, P.: Problem of Incompleteness in Textual Requirements Specification. In: *Proceedings of the 14th International Conference on Software Technologies*, vol. 1, pp. 323–330. INSTICC, SCITEPRESS – Science and Technology Publications, Porto, Portugal (2019). DOI 10.5220/0007978003230330
23. Šenkýř, D., Kroha, P.: Patterns for Checking Incompleteness of Scenarios in Textual Requirements Specification. In: *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, vol. 1, pp. 289–296. INSTICC, SCITEPRESS – Science and Technology Publications, Porto, Portugal (2020). DOI 10.5220/0009344202890296
24. Šenkýř, D., Kroha, P.: Problem of Inconsistency and Default Consistency Rules. In: H. Fujita, H. Pérez-Meana (eds.) *New Trends in Intelligent Software Methodologies, Tools and Techniques – Proceedings of the 20th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques, SoMeT 2021, Cancun, Mexico, 21–23 September, 2021, Frontiers in Artificial Intelligence and Applications*, vol. 337, pp. 674–687. IOS Press (2021). DOI 10.3233/FAIA210063
25. Suchánek, M., Mannaert, H., Uhnák, P., Pergl, R.: Bi-directional Transformation between Normalized Systems Elements and Domain Ontologies in OWL. In: R. Ali, H. Kaindl, L.A. Maciaszek (eds.) *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2020, Prague, Czech Republic, May 5–6, 2020*, pp. 74–85. SCITEPRESS (2020). DOI 10.5220/0009356800740085
26. Suchánek, M., Pergl, R.: Towards Evolvable Documents with a Conceptualization-Based Case Study. *International Journal on Advances in Intelligent Systems* **11**, 212–223 (2018)
27. Talele, P., Phalnikar, R.: Software Requirements Classification and Prioritisation Using Machine Learning. In: A. Joshi, M. Khosravy, N. Gupta (eds.) *Machine Learning for Predictive Analysis*, pp. 257–267. Springer Singapore, Singapore (2021)
28. van Rooijen, L., Bäumer, F.S., Platenius, M.C., Geierhos, M., Hamann, H., Engels, G.: From User Demand to Software Service: Using Machine Learning to Automate the Requirements Specification Process. In: 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), pp. 379–385 (2017). DOI 10.1109/REW.2017.26
29. Šenkýř, D., Kroha, P.: Problem of Inconsistency in Textual Requirements Specification. In: R. Ali, H. Kaindl, L.A. Maciaszek (eds.) *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering – ENASE*, pp. 213–220. INSTICC, SciTePress (2021). DOI 10.5220/0010421602130220
30. Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K.J., Ajagbe, M.A., Chioasca, E.V., Batistana-Navarro, R.T.: Natural Language Processing (NLP) for Requirements Engineering: A Systematic Mapping Study (2020)