Detecting Ambiguities in Requirements Documents Using Inspections

Erik Kamsties*, Daniel M. Berry**, Barbara Paech*

*Fraunhofer Institute for
Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
{kamsties, paech}@iese.fhg.de

**Department of Computer Science
University of Waterloo
200 University Ave. West
Waterloo ON, N2L 3G1, Canada
dberry@csg.uwaterloo.ca

June, 2001

Abstract

Natural language is the most used representation for stating requirements on computer-based systems in industry. However, natural language is inherently ambiguous. Customers and software developers can disagree on the interpretation of a requirement without being aware of this fact. This disagreement can result in disastrous software failures.

We argue that ambiguity is a more complex phenomenon than is often recognized in the literature. While inconsistencies and some types of incompleteness can be mechanically detected in formal specifications, ambiguities in informal specicifications often result in incorrect formal specifications. These misinterpretations can be detected only by execution or inspection of the formal specification. We suggest an inspection technique for detecting ambiguities in informal requirement documents before formal specifications are produced in order to avoid misinterpretations, rather than searching for them afterwards. We show how this technique can be tailored to different project contexts with the help of metamodels to increase its effectiveness. Finally, we report on experimental validation of the technique.

1 Introduction

In industrial requirements engineering (RE), natural language is the most frequently used representation in which to state requirements that are to be met by information technology products or services. Natural language is universal, flexible, widespread, but unfortunately also inherently ambiguous. Words can have several meanings. For instance, the Oxford English Dictionary says that the 500 most used words in English have on average 23 meanings.§ Also, phrases and whole sentences can be interpreted in more than one way.

Ambiguous requirements are a serious problem in software development, because often stakeholders are not even aware that there is an ambiguity. Each gets from reading the requirements an understanding that differs from that of others, without recognizing this difference. Consequently, the software developers design and implement a system that does not behave as intended by the users, but the developers honestly believe they have followed the requirements. Also, components fail to interact properly, because the same requirement was allocated to different components, and the different developers of the components have interpreted the requirement differently.

We distinguish *linguistic* and *RE-specific* ambiguity. The former is context independent and can be observed by any reader who has a tone for language. An example is

(1) The product shall show the weather for the next 24 hours.

The phrase for the next twenty-four hours can be attached to the verb show or to the noun weather. Thus, the requirement can be interpreted as the product shall show the current weather and continue to do so for the next 24 hours or the product shall show the projected weather for the forthcoming 24 hours.

An RE-specific ambiguity is context dependent and can be observed only by a reader who has knowledge of the particular requirements context or of the other requirements. Parnas, Asmis, and Madey give an example of such an ambiguity in a requirement about a continually varying water level in a tank: [23]

(2) Shut off the pumps if the water level remains above 100 meters for more than 4 seconds.

They claim that this type of ambiguity is very common in informal requirements documents. One can find four interpretations:

§ Cited after Jeff Gray, http://www.vuse.vanderbilt.edu/~jgray/ambig.html

1. Shut off the pumps if the *mean* water level over the past 4 seconds was above 100 meters.
2. Shut off the pumps if the *median* water level over the past 4 seconds was above 100 meters.
3. Shut off the pumps if the *root mean square* water level over the past 4 seconds was above 100 meters.
4. Shut off the pumps if the *minimum* water level over the past 4 seconds was above 100 meters.

However, the software engineers did not notice this ambiguity and quietly assumed the fourth interpretation. Unfortunately, under this interpretation, with sizable rapid waves in the tank, the water level can be dangerously high without triggering the shut off. In general, the interpretation of the ambiguity is very much a function of the reader's background. For example, in many other engineering areas, the standard interpretation would be the third.

RE-specific ambiguities are more important than linguistic ones. Although a requirements sentence may be ambiguous because of multiple word senses, syntactic sentence readings, or referenced items, psycho-linguistic experiments show that there is often one preferred sentence reading after semantics and the context are considered [24]. In the requirements documents that we have investigated, RE-specific ambiguities account for the majority of ambiguities, while purely linguistic ambiguities played a less significant role. For one requirements document, 4 linguistic but 54 RE-specific ambiguities were reported [19].

We propose in this paper an inspection technique for detecting ambiguities that is based on a checklist and scenario-based reading [3]. Checklists are widely applied in industry, and scenario-based reading has been demonstrated in industry as an effective method to defect detection [3]. The focus of our inspection technique is on RE-specific ambiguities.

Section 2 presents related work, and Section 3 distills an inspection approach from the related work. Section 4 introduces a new understanding of ambiguity to increase the awareness of them by inspectors. Then, Section 5 presents an inspection technique for ambiguities. Next, Section 6 sketches one possible approach to tailor inspection techniques to a specific RE context. Section 7 describes experimental validation of the effectiveness of the technique and of the tailoring approach. Finally, Section 8 concludes the paper.

This paper is based on research conducted by the first author under the other authors' supervision. This work is described in the first author's Ph.D. dissertation, titled *Surfacing Ambiguity in Natural Language Requirements* [19]. Due to space limitations, this paper presents only one technique and tailoring to only one specific RE context. The full dissertation describes several techniques and tailorings to several contexts. It describes experimental validation for all the techniques and tailorings and ranks them by their effectiveness. The reader is urged to consult the dissertation for these additional details.

2 Related Work

The most recommended solution to the ambiguity problem is the use of a formal requirements specification languages, such as SCR [13], or a semi-formal requirements specification languages, such as UML [1], rather than natural language. Such a language has a more-or-less well-defined semantics. Thus, the degree of ambiguity in requirements is at least significantly diminished if not eliminated. However, a requirements specification language can solve the problem only if all stakeholders can read and write requirements using the language. Usually, the more formal the language, the fewer the stakeholders that can read and write the language. Moreover, even when such a language is used, the initial requirements are written in natural language. These initial requirements must be translated into the formal or semi-formal language by a requirements engineer. During this translation, an ambiguous informal requirement is often not recognized as such, and it ends up becoming an unambiguously wrong formal or semi-formal requirement, if it is unconsciously misinterpreted. Such a misinterpretation can slip through undetected, because of stakeholders' aversion to reading requirements written in a formal or semi-formal language. Simulation, another way of validating formal requirements, can show only the presence of misinterpretations but not their absence.

Ambiguities can be detected by also a Natural Language Processing (NLP) tool. However, current NLP tools suffer from several problems. First, often they require restricting the syntax of natural language requirements. Second, they require expert programming to be made able to parse arbitrary text [11]. Third, they raise many more ambiguities than are really perceived by a human. Some tools try to prevent this problem by making a default interpretation [9], but then, the task of detecting ambiguities is shifted to the tool user, as the tool effectively unconsciously misinterprets.

Several inspection techniques have been proposed for spotting ambiguities. See, for example, references [22, 10, 8, 26]. Perhaps, the most effective approach is to hand requirements to several different stakeholders, to ask each for an interpretation, and to compare these interpretations afterwards. If the interpretations differ, the

requirements are ambiguous [10]. Obviously, this approach is feasible only for small sets of requirements. A detailed checklist of ambiguous words often used in requirements is provided in reference [8], and a checklist derived from Neuro-Linguistic Programming is provided in reference [26]. However, these checklists do not address RE-specific ambiguities. Most other inspection techniques assume that inspectors are able to detect ambiguities just by reading; no guidance is provided on how to find an ambiguity. There is usually one checklist item asking, "is the requirement ambiguous?" The major problem of ambiguity is not being aware of it. Thus, simply asking whether there is an ambiguity is not much help.

An ambiguous requirement is often defined in the RE literature as a requirement that has more than one interpretation. Terms, pronoun references, and certain sentence structures are shown to be sources of ambiguity [25]. Occasionally, the broader RE context behind the written requirements has been recognized as a source of ambiguity [27]. Also, it has been recognized that the RE context can help to disambiguate a requirement and that a certain amount of contextual knowledge is required from the reader; otherwise every requirement appears ambiguous [25, 7].

3 Inspection Approaches to Detecting Ambiguity

Based on the past work in ambiguity detection, we realized that an inspection approach had good potential. Certainly, checklists can be developed for kinds of ambiguities identifiable up front, such as linguistic and some of the RE-specific ambiguity types. However there are RE-specific ambiguities that depend on the model (Examples are given in Sections 3 and 5) used to specify the requirements. Building a formal or semi-formal model does eliminate ambiguities by causing, often unconscious, chosing of one interpretation. Thus, modeling, of and by itself, does not really solve the ambiguity problem, that is, of *recognizing* ambiguous requirements so that they can be disambiguated correctly. However, the fact that the *process* of building a formal or semi-formal model requires disambiguation means that the process can be used to our advantage. That is, as we build the formal or semi-formal model, we watch for ambiguities with heightened awareness of the phenomenon. More than that, we do not really have to build a model, so long as we ask the ambiguity-exposing questions that we *would* ask while building the model. Therefore, we enhance the traditional check-list directed inspection process by questions identified during a modeling process. The questions depend on the modeling language used, and are thus based on the modeling language's metamodel. This enhancement is called *tailoring by a metamodel*. This paper shows fragments of tailoring by UML.

4 A New Understanding of Ambiguity

This section provides a new understanding of ambiguity in RE that is influenced by linguistics. An improved understanding of possible defects has two effects:

1. It helps find otherwise undetected defects in requirements documents through improved awareness, and
2. it helps improve checklists for requirements validation [21].

We define a requirement as ambiguous if *it has multiple interpretations despite the reader's knowledge of the RE context*. It does not matter whether the author unintentionally introduced the ambiguity, but knows what was meant, or he or she intentionally introduced the ambiguity to include all possible interpretations. This definition is useful in the context of inspections, because at the time a requirements document is read by an inspector, it is difficult if not impossible to distinguish intentional-but-possibly-acceptable ambiguity from unintentional-and-unacceptable ambiguity.

We need a more comprehensive list of possible types of ambiguity in requirements and a more precise understanding of the RE context than are found in the current RE literature. For this purpose, we have investigated the understanding of ambiguity in linguistics, e.g., in reference [20], and in natural language processing, e.g., in reference [2]. Our understanding of the RE context is inspired by the WRSPM (World, Requirements, Specifications, Program, and Machine) model [12] and by the Four-World model [16], which structure the RE context:

The *RE context* that must be considered when interpreting a requirement consists of the

- *requirements document* of which the considered requirement is part;
- *application domain*, e.g., organizational environment and behaviors of external agents;
- *system domain*, e.g., conceptual models of software systems and their behavior; and
- *development domain*, e.g., conceptual models of development products and processes.

Also, real-world knowledge and language knowledge help to disambiguate an ambiguity. However, we assume that this knowledge is sufficiently shared by requirements authors and readers, and thus, it is not included in the RE context. Ambiguity can arise also from the social, cultural, political, or personal context, but this sort of ambiguity is outside the scope of this work.

In the remainder of this section, the only types of ambiguity discussed are those that are not mentioned in the RE literature or are not discussed comprehensively enough. A complete discussion can be found in [17, 19].

Polysemy occurs when a word has several related meanings, e.g., green. In contrast, a word is homonymous when it has unrelated meanings, e.g., bank. In the context of the example,

(3) When the user inserts the paper strip, the Tamagotchi is set to its defaults.,

the word Tamagotchi is used both as the name of a toy, i.e., an electro-mechanical device, as well as a creature simulated by this toy. Thus, Requirement 3 can mean that the whole toy or just the creature can be set to its defaults. Polysemies are a much larger problem in requirements documents than are homonyms. The meanings of a polysemy are related. Thus, more detailed contextual information is necessary to disambiguate it than, for instance, in the case of bank.

Systematic polysemy applies to a class of words. The *volatile–persistent ambiguity*, for instance, arises when a word of a requirement refers to either a volatile or a persistent property of an object. In the requirement,

(4) When the user presses the L- and R-button simultaneously, the alarm is turned off.,

the phrase turned off can refer to an alarm that is currently sounded by the system or to the general ability of the system to raise alarms.

Scope ambiguity occurs when quantifiers, e.g., every, each, all, some, several, a, and negations, e.g., not, enter into different scope relations with the other scoped parts of the requirements sentence. In other words, the ambiguity lies in the precedence of these operators. Quantifiers are discussed in more detail in references [4, 5].

(5) **All** sections have **a** hallway.

The quantifier a can take wide scope over sections, i.e., all sections share a hallway, or it can take narrow scope, i.e., each section has a hallway.

Referential ambiguity is caused by an anaphor in a requirement that refers to more than one element introduced earlier in the sentence or in a sentence before. Anaphora include not only the well-known pronouns, e.g., it, they, but also definite noun phrases and elliptical expressions.

(6) ... The product shall show **the roads** predicted to freeze.

The definite noun phrase the roads can refer to more than one set of roads that are specified earlier in the requirements document.

(7) If.... If the ATM accepts the card, the user enters the PIN. If **not**, the card is rejected.

The word not is here an elliptical expression that refers either to the condition specified in the previous sentence or to something written before.

Discourse ambiguity occurs when the relation of one requirement to other requirements in the requirements document is ambiguous, but not because of anaphora.

(8) When the user pulls the paper strip, the cyber chicken is born. ... After its first night, the cyber chicken becomes a Marutchi. ... After 2 days, the cyber chicken becomes a Tamatchi (friendly teen).

The phrase After 2 days refers to a contextually specified event; two days after this event, the cyber chicken changes its state. Since there are more than one event described, either of which can be referenced by the phrase After 2 days, the requirement is ambiguous.

(9) The user can play with the cyber chicken. A game is started by choosing the menu item “Play”. The display shows a playing chicken. A game has five rounds and can be won or lost. One game follows another; a game can be aborted by the R-button. ... The user must **play** at least twice a day to develop the Tamatchi character of the cyber chicken.

Ignore the fact that the idea of the game is not clear from these sentences. The verb play refers to the process of playing with the cyber chicken. This process can take several directions: A game can be won or lost; a game can be aborted immediately after being started; and so on. Therefore, the meaning of play is ambiguous.

Application-domain ambiguity occurs because a requirement allows several interpretations with respect to what is known about the application domain. It may not obvious that knowledge about a domain, e.g., the application domain, can cause ambiguity. Consider the requirement

(10) Generate a dial tone.

This requirement is ambiguous in international telecommunication systems due to conflicting national standards about dial tone generation; in telecommunication systems within any one country, the requirement is unambiguous. Another example of application-domain ambiguity is Requirement 2, discussed in the introduction.

System-domain ambiguity occurs because a requirement allows several interpretations with respect to what is known about the system domain.

(11) If the timer expires before receipt of a [T-DISCONNECT indication], the SPM requests [transport disconnection] with a [T-DISCONNECT request]. The timer is cancelled on receipt of a [T-DISCONNECT indication].

It is ambiguous whether or not the second sentence is part of the conditional in the first sentence. This particular requirement could be disambiguated by RE-context knowledge; the cancellation of an expired timer probably makes little sense.

Development-domain ambiguity occurs because a requirement allows several interpretations with respect to what is known about the development domain.

(12) The doors of the lift never open at a floor unless the lift is stationary at that floor.

The statement can be interpreted either as an environmental property describing a mechanical lock that prevents the doors from opening or as a requirement that must be implemented by the software developer. That is, the statement can be interpreted as either indicative or optative [14, 15]. We classify this mood ambiguity as a development-domain ambiguity, because it is unambiguous relative to the application and system domain. However, it remaims open as to whether the statement is a requirement to be implemented in the software or the statement can be assumed as already provided by the hardware. This type of ambiguity occurs frequently in German requirements documents. In U.S. requirements documents, the word "shall" is often used to identify requirements, in the optative mood, reserving the word "will" for statements, in the indicative mood, that are true about the environment.

Scope and referential ambiguity are linguistic ambiguities. The other kinds of ambiguity are RE specific. Since a polysemy is context dependent in most cases, it may very well be the case that an ambiguity can be classified as both a polysemy and a application-, system-, or development-domain ambiguity. The ambiguity in Requirement 2 could be classified as polysemy or system domain ambiguity.

5 Ambiguity Detection Technique

In order to provide an ambiguity detection technique that is applicable in industry, we improve well-accepted inspection-based RE techniques rather than develop revolutionary new ones. We have selected two techniques:

1. checklists and
2. scenario-based reading [3].

We chose checklists, because they are perhaps the most widely applied technique for detecting defects in requirements documents. A checklist is useful for checking single requirements, but less useful for finding discourse ambiguities, if these occur between requirements that are scattered over several pages. Thus, we have selected also scenario-based reading.

The overall idea of scenario-based reading is to provide an inspector with an operational scenario, which requires him or her to first create an abstraction of a product, and then to answer questions based on analyzing the abstraction with a particular emphasis or role that the inspector assumes. For example, the inspector might create test cases for a requirements document and then answer the question, "Do you have all information necessary to develop a test case?". If the question cannot be answered, then a defect may have been detected.

The previously identified ambiguity types can be mapped easily into a checklist. We recommend creating a separate checklist for ambiguity and putting important RE-specific ambiguity types into the list. In our experience, linguistic ambiguities, except for lexical and referential ambiguity, can usually be resolved by the reader. Table 1 shows a basic ambiguity checklist.

Table 2 shows a discourse-ambiguity-spotting scenario that we have developed to make use of *interaction matrices*. Interaction matrices were suggested originally to reveal requirements overlaps and conflicts [28]. We

have changed the format of the interaction matrix slightly. Instead of a matrix, we use a two-column table. The first column of a row contains the ID of the considered requirement and the second column contains the IDs of the requirements that are related to the considered requirement, as shown in Table 4. Possible relations between two

Item	Description
Lexical Ambiguity, Polysemy	Does a word in a requirement have several possibly related meanings? Be aware that *lexical ambiguity* arises in particular from the actual usage of a word in an RE context.
Systematic Polysemy	A systematic polysemy applies to a class of words: (1) The *object–class ambiguity* arises when a word in a requirement can refer either to a class of objects or to just a particular object of the same class. (2) The *process–product ambiguity* arises when a word can refer either to a process or to a product of the process. (3) The *volatile–persistent ambiguity* arises when a word refers to either a volatile or a persistent property of an object.
Referential Ambiguity	Can a phrase in a requirement refer to more than one object in other requirements? Check pronouns (it), definite noun phrases (the roads), and ellipses (... If not, ...).
Discourse Ambiguity	Does a requirement have several interpretations in relation to other requirements? This ambiguity arises when (1) words such as first, before, between, after, and last are used and can refer to several elements and when (2) adjectives, verbs, or noun phrases refer to more than one condition described before.
Domain Ambiguity	Is the requirement ambiguous with respect to what is known about the application, system, or development domain?

Table 1. Checklist for Ambiguity Detection

Interaction Table Scenario
Create an interaction table for the requirements document to record the relations among the requirements by using the provided form. The goal is to identify ambiguities in the relation of one requirement to other requirements. Follow the procedure below to create the interaction table and answer the provided questions to identify ambiguities.
Create the interaction table For each requirement, identify the related requirements. Related requirements are requirements that describe some condition, event, or process, which is a prerequisite for, affects, or supplements in some way the considered requirement. Record the ID of the considered requirement in Column 1 and the IDs of the related requirements in Column 2 of the form. For each related requirement, provide a short description of the nature of this relation in Column 2 in addition to the ID. **Questions:** – Are you uncertain whether or not a requirement *A* is related to a requirement *B*? – If a requirement *A* is related to another requirement *B*, can this target requirement *B* be identified unambiguously, or are there several possible target requirements? – If a requirement *A* is unambiguously related to a requirement *B*, is the type of this relation clear, or are several types of relations possible?

Table 2. Scenario for Detection of Discourse Ambiguity

requirements *A* and *B* include:

- *A* overlaps with *B*,
- *A* conflicts with *B*,
- *A* adds detail to *B*, i.e., *A* specifies *B*,
- *A* requires *B*, and
- *A* constrains *B*.

An ambiguity is detected when either the type of relation between two requirements *A* and *B* or the targeted requirement *B* of a relation allows several interpretations.

We look at an example of the use of the scenario to identify potential ambiguities. The Engineered Safety Feature Actuation System (ESFAS), used in nuclear power plants, prevents or mitigates damage to the core and coolant system when a fault, such as loss of coolant, occurs [6]. The ESFAS monitors a pressure sensor and determines whether an actuation signal called safety injection must be sent to the safety feature components that cope

with pressure accidents. The human operator has to press two momentary pushbuttons to block the signal and to reset the blockage. A part of the ESFAS requirements is shown in Table 3.

R1	The system monitors the pressure and sends the safety injection signal when the pressurizer's pressure falls below a 'low' threshold.
R2	The human operator can override system actions by *pushing on a 'Block' button* and resets the *manual block* by pushing on a 'Reset' button.
R3	A *manual block* is permitted if and only if the pressure is below a 'permit' threshold.

Table 3. Part of ESFAS Requirements

The application of the interaction table scenario leads to the interaction table shown in Table 4.

Req.	**Related Requirements**
R1	R1 is constrained by R2: Sends the safety injection signal in R1 is a system action that can be overridden by the human action described in R2.
R2	R2 is constrained by R3: However, it is ambiguous what exactly is constrained, the event of pushing on a 'Block' button or the duration of the system state 'manual block'.
...	...

Table 4. Interaction Table for ESFAS

The development of the interaction table leads to the detection of a discourse ambiguity in R3 concerning the phrase manual block. With respect to R2, the ambiguity lies in whether the event Block button pushed or the system state manual block or both is meant in R3.

6 Identification of Ambiguity Types

The checklist provided in the last section describes the RE-specific ambiguity types rather abstractly, because they depend on the particular RE context in which the RE process takes place. A checklist and a scenario that are more effective in this respect can be developed if a *metamodel* is available that characterizes the particular RE context. Such a metamodel allows identifying ambiguity types that are typical for this RE context, as we show in this section.

A metamodel defines a language for specifying a model. The metamodel can be available from previous metamodeling efforts, e.g., by the NATURE project [16], and from the employed modeling languages, e.g., the UML has a published metamodel [1].

An ambiguity in a natural language requirement leads to different models. Therefore, the idea is to identify and describe types of ambiguity from available metamodels. As illustrated in Figure 1, the customers and users speak a language different from that of the requirements engineers and developers. A natural language requirement r_i in the customers' language is ambiguous if it can be translated into at least two different models m_{i1}, $\cdots$, and m_{in} in the requirements engineers' language. For example, the following requirement contains at least two ambiguities,

(13) Aircraft that are non-friendly and have an unknown mission or the potential to enter restricted airspace within 5 minutes shall raise an alert.,

and can be translated into different UML models, two of which are shown in Figure 2. First, there is a linguistic ambiguity that concerns and and or, because we cannot assume priority rules of Boolean logic for natural language. In the left state diagram, or precedes and, and in the right state diagram, and precedes or. Second, there is a RE-specific ambiguity, that concerns the alert. UML distinguishes actions from activities. An action is executed when a transition fires, a state is entered, or a state is exited. An activity is an action that is performed while being in a state. The phrase raise an alert can be interpreted as an action, as in the left state diagram, or as an activity, as in the right state diagram.

From the viewpoint of a metamodel *mm*, the models differ in the subsets, mm_{j1}, mm_{j2}, $\cdots$, and mm_{jn}, of *mm* that are employed. That is, the models differ in the employed concept types, relationships among concept types, attribute values of instantiated concept types, or links among instantiated concept types. For example, the two models depicted in Figure 2 make use of the same concept types, i.e., state, transition, and action, depicted in Figure

5, but their relationships differ. In the left model, the action plays the role of an effect in relation to the transition. In the right model, the action plays the role of a do–activity in relation to the state.

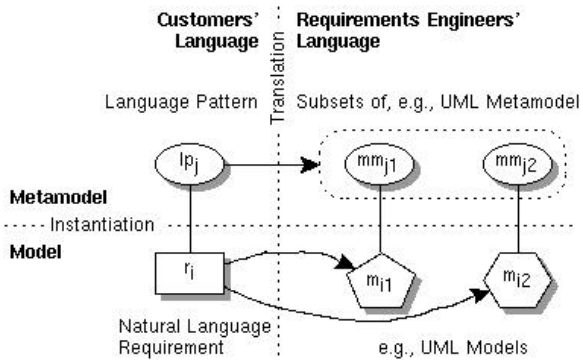

Figure 1. Role of the Metamodel

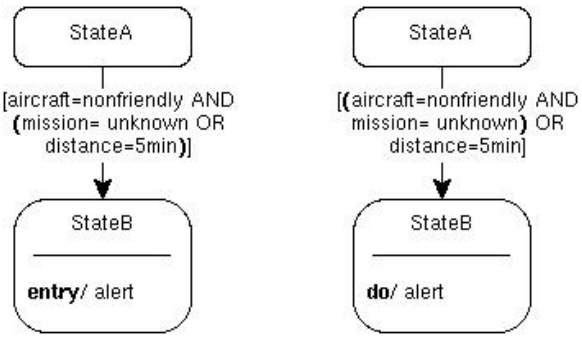

Figure 2. UML State Diagrams for Requirement 13

We have developed a set of heuristics to help a requirements engineer to systematically identify subsets mm_{j1}, mm_{j2}, $\cdots$, and mm_{jn} of a metamodel *mm* that could be subject to ambiguity. The heuristics can be implemented by a tool. The essentially manual task that is left is the search for ambiguous language patterns lp_j that can be translated into more than one of the identified subsets. Already known patterns should be investigated and new patterns show up from analyzing example requirements. Since there is no metamodel of unbounded natural language, the results of searching for ambiguous language patterns depend on the language skills of the requirements engineer.

Let us remind the reader that investigation of a metamodel helps to identify types of ambiguity in order to improve a reading technique. The actual translation of informal requirements into some formal or semi-formal representation is not necessary to spot ambiguities. However, the decisions between alternative interpretations of an informal requirement, which are often made unconsciously during modeling, are now explicitly incorporated into an existing inspection approach.

We use the UML metamodel to illustrate the heuristics, which is documented also in UML. We remind the reader that a class denotes a *concept type*, an association denotes a *relationship* between several concept types, and inheritance denotes a *specialization* of a concept type. The heuristics can be applied also to other metamodels documented in other languages than UML. The UML metamodel contains background information about the system domain. Thus, the investigation of the UML metamodel leads to the detection of potential types of system domain ambiguity.

The set of heuristics allows the exhaustive and thus systematic investigation of a metamodel. Each heuristic focuses on a different language construct of the metamodel, which is reflected in the name of the heuristic. There are four heuristics: *Analyze Specializations*, *Analyze Relationships*, *Analyze Constraints*, and *Analyze Concept Types*. The heuristics together cover all language constructs of the metamodel. That is, they cover the meta-metamodel. Note that only those concept types, attributes, and relationships that are relevant to RE need to be investigated.

The more specific the metamodel, the more specific the types of ambiguity that result from the application of the heuristics. Some metamodels concentrate on a small set of very generic concepts, while others offer a large set of specific concepts to capture as much semantics as possible. A generic metamodel can be used to describe all kinds of information in the application, system, and development domain. A specific metamodel is tailored to a particular modeling purpose within a particular domain. For example, the metamodel of Entity–Relationship diagrams is generic and the diagrams can be used to describe information in all domains. On the other hand, the metamodel of SCR, which we extracted from [13], is specific. An SCR model can be used as a specification model only in the system domain. The subsections below present three of these heuristics. Space limitations prevent us from presenting the heuristic concerning constraints. For more details, see reference [19].

6.1 Analyze Specializations

Specialization is used in a metamodel to describe similar concept types. A topmost, typically abstract, concept type captures their commonalities. Their differences are captured in concept types that are derived by specialization from the common concept type. We should check for ambiguous language patterns,

- if the metamodel contains a specialization hierarchy and there is more than one non-abstract concept type in the specialization hierarchy.

Figure 3 shows a part of the UML metamodel. The specialization hierarchy of ModelElement comprises several non-abstract concept types including Attribute, Class, and Association. There are several ambiguous language patterns; nouns describe classes or attributes; verbs describe attributes or associations.

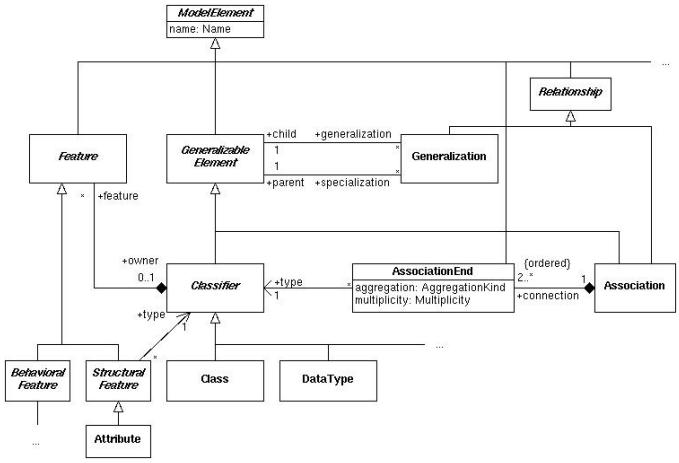

Figure 3. Excerpt of UML Foundation Package

Model–element ambiguity arises from a verb or noun when the verb or noun that describes a model element can be interpreted in more than one way, i.e., as

1. an attribute,
2. a class,
3. a data type,
4. a generalization, or
5. an association.

Model–element ambiguity is a type of system-domain ambiguity, and it occurs often with respect to attributes and classes as a kind of design issue. A noun can be translated often as an attribute as well as a class.

(14) Each person has a social security number.

The requirement can be interpreted either as an association between two classes, person and social security number, or as a class person consisting of an attribute social security number, as shown in Figure 4.

Figure 4. Class Diagrams for Requirement 14

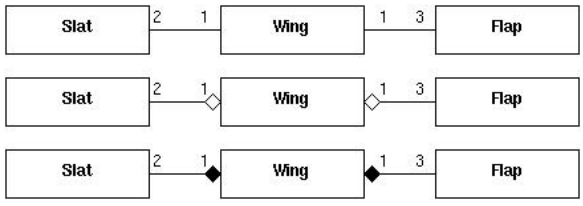

Figure 6. Class Diagrams for Requirement 15

6.2 Analyze Relationships

Relationships between concept types describe how concept types must be combined in a well-formed, i.e., meaningful, model. For example, the relationships in the metamodel shown in Figure 5 require each transition in a UML model to be connected with exactly one source state and one target state. Relationships are sometimes optional and, thus, allow more than one way to combine the same concept types, each of course, with slightly different semantics. We should check for ambiguous language patterns,

- if two concept types are connected by two or more relationships, a concept type plays in each relationship a different role, and the relationships are optional, i.e., the multiplicity of each includes zero, or
- if more than two concept types are related by optional relationships so that different combinations of the same set of concept types are possible.

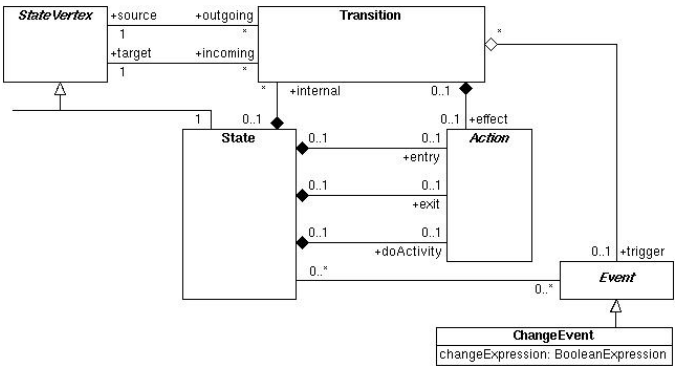

Figure 5. Excerpt of UML State Machine Package

The heuristic can be applied thrice on the part of the state machine package depicted in Figure 5. First, Transition can play the role of an incoming or outgoing Transition with respect to StateVertex. Second, there are four ways to combine StateVertex, Transition, State, and Action. Action can be associated as an entry, exit, or doActivity to State or it can be associated as an effect to Transition. Moreover, Transition can be associated as an internal Transition to State or it can be external to the involved States. We focus on the third case, i.e, the multiple roles of Action in relation to State.

Action ambiguity is a type of system-domain ambiguity, and it arises when a verb that describes an action of a system can be interpreted as an action that is executed in more than one way, i.e.,

1. when a state is entered,
2. when a state is exited, or
3. while being in a state.

An example of an action ambiguity is Requirement 13.

6.3 Analyze Concept Types

A concept type represents a language construct of the language that is defined by the metamodel. A concept type or a set of concept types can be instantiated in more than one way. Moreover, an attribute is sometimes used in a metamodel as a lightweight way to define variants of concept types. We should check for ambiguous language patterns,

- if an attribute of a concept type defines variants of that concept type or
- if a concept type is concrete and can be instantiated in several ways that differ in attribute values or links to other instantiated concept types.

Consider the UML Foundation package shown in Figure 3. AssociationEnd has two attributes, aggregation and multiplicity. Aggregation defines variants of the concept type, namely no aggregate, aggregate, and strong aggregate (i.e., composite). Multiplicity can have several attribute values. Moreover, AssociationEnd can be instantiated in several ways so that the linked instances of Classifier differ. We focus on the attribute aggregation.

Aggregation–kind ambiguity arises when a verb phrase, such as has, is part of, or consists of, can be interpreted in more than one way as an association between two classifiers, i.e.,:

1. the two classifiers are related but are not aggregates of each other,
2. one classifier is an aggregate of the other classifier, or
3. one classifier is composed of, i.e., strongly owns, the other classifier, and this other classifier may not be part of any other composite.

In the requirement,

(15) Each aircraft wing has two slats and three flaps.

Slats and Flaps can be strongly owned by a Wing, parts of a Wing, or can be just somehow related to a Wing. Requirement 15 suffers also from model–element ambiguity; Slat and Flap could also be attributes of Wing.

6.4 Building an Ambiguity Detection Technique

Using these heuristics, we developed a second ambiguity detection technique geared specially for the embedded-systems domain. See reference [19] for the full details about the technique.

7 Validation

Space limitations prevent us from describing the full details of the validation, as they involve carrying out the heuristics to two metamodels and then a series of controlled experiments to measure the effectiveness of various ambiguity detection techniques. More details are found in reference [18], and full details are found in reference [19]. Here, we describe only the major conclusions. Do note that results reported here were determined to be statistically significant; that is, the experiments were controlled experiments carried out with sufficiently large numbers of subjects and with designs that minimized threats to validity.

We applied the heuristics to a subset of the UML Foundation package and to the SCR metamodel. We identified 7 types of ambiguity in the investigated subset of the UML Foundation package and 11 in the SCR metamodel. The complexities of these metamodels were such that it took from 1 to 3 days to investigate one metamodel. Analyzing concept types proved to be the most effective heuristic in terms of the number of identified ambiguity types.

We performed several controlled experiments about ambiguity detection techniques with student subjects at two universities in Germany.

- The result of one case study is that the inspection techniques suggested in this paper lead to the detection of more ambiguities (18-25%) than when a requirements specification language is used to develop a requirements model from the same requirements document (9%). This result supports the idea behind the techniques' providing the reviewer with more prescriptive guidance on how to spot ambiguities, including descriptions of possible types of linguistic and RE-specific ambiguities.
- The result from several controlled experiments is that a domain-specific ambiguity detection technique, which is developed on the basis of a metamodel, is more effective and efficient (25% of the ambiguities are detected, and in the same time) than the generic technique presented in Section 4 (18% of the ambiguities are detected). This result confirms that an ambiguity detection technique needs to be tailored to an RE context in order to be most effective.
- The experiments show also that one cannot expect to find all ambiguities in a requirements document with realistic resources. Given a team of three reviewers that spends a total of 4.5 hours, 18% of the ambiguities were detected if the presented technique was applied, and 25% of the ambiguities were detected with a domain-specific technique. However, there is no need to detect all ambiguities; the experiment with requirements specification languages has shown that 72% of the ambiguities were interpreted correctly.

8 Conclusions and Future Work

This paper makes two contributions to reduce the level of ambiguity in industrial requirements documents. First, it offers to a requirements engineer an efficient inspection technique for detecting ambiguous requirements that is applicable in industrial RE. Second, it offers an approach to identify ambiguity types that can occur in a particular RE context. Realistically, one cannot expect to identify types of ambiguity that he or she never ever has thought about or come across. Rather, the contribution lies in the systematic way to explore this implicitly existing knowledge by using the heuristics and in increasing the requirements engineer's awareness of the problem.

Our future work aims at investigating in how much meetings increase the number of detected ambiguities. In meetings, perhaps ambiguities that slipped through individual preparation can be detected. We must determine which meeting formats allow reviewers to best exchange their interpretations of requirements.

Acknowledgements

The authors would like to thank the students who participated in the experiments and the colleagues who helped in conducting them, in particular Antje von Knethen, Jan Philipps, and Bernhard Schaetz. Berry was supported in parts by a University of Waterloo Startup Grant and by NSERC grant NSERC-RGPIN227055-00.

References

[1] "OMG Unified Modeling Language," Technical Report, Version 1.3, Object Management Group (June 1999).

[2] Allen, J., *Natural Language Understanding,* Addison-Wesley, Reading, MA (1995), Second Edition..

[3] Basili, V.R., "Evolving and Packaging Reading Technologies," *Journal of Systems and Software* **38**(1), pp. 3–12 (1997).

[4] Berry, D.M. and Kamsties, E., "The Dangerous 'All' in Specifications," pp. 191–194 in *Proceedings of 10th International Workshop on Software Specification & Design, IWSSD-10*, IEEE CS Press (2000).

[5] Berry, D.M., Kamsties, E., Kay, D.G., and Krieger, M.M., "From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity," Technical Report, University of Waterloo, Waterloo, ON, Canada (2001).

[6] Courtois, P.-J. and Parnas, D.L., "Documentation for Safety Critical Software," pp. 315–323 in *Proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD (May 1993).

[7] Davis, A., Overmyer, S., Jordan, K., *et al*, "Identifying and Measuring Quality in a Software Requirements Specification," pp. 141–152 in *Proceedings of METRICS '93*, Baltimore, MD (May 1993).

[8] Freedman, D.P. and Weinberg, G.M., *Handbook of Walkthroughs Inspections and Technical Reviews,* Dorset House, New York, NY (1990).

[9] Fuchs, N.E. and Schwitter, R., "Attempto Controlled English (ACE)," in *Proceedings of the First International Workshop on Controlled Language Applications (CLAW'96)*, Belgium (March 1996).

[10] Gause, D.C. and Weinberg, G.M., *Exploring Requirements: Quality Before Design,* Dorset House, New York, NY (1989).

[11] Gervasi, V. and Nuseibeh, B., "Lightweight Validation of Natural Language Requirements," pp. 140–148 in *Proceedings of Fourth IEEE International Conference on Requirements Engineering (ICRE'2000),*, Schaumburg, IL (19–23 June 2000).

[12] Gunter, C.A., Gunter, E.L., Jackson, M., and Zave, P., "A Reference Model for Requirements and Specifications," *IEEE Software* **17**(3), pp. 37–43 (May/June 2000).

[13] Heitmeyer, C.L., Jeffords, R.D., and Labaw, B.G., "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Software Engineering and Methodology* **5**(3), pp. 231–261 (July 1996).

[14] Jackson, M. and Zave, P., "Domain Descriptions," pp. 56–64 in *Proceedings of the International Symposium on Requirements Engineering*, IEEE Computer Society (1993).

[15] Jackson, M., *Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices,* Addison-Wesley, London (1995).

[16] Jarke, M., Rolland, C., Sutcliffe, A., and Dömges, R., *The NATURE of Requirements Engineering,* Shaker Verlag, Aachen, Germany (1999).

[17] Kamsties, E. and Paech, B., "Taming Ambiguity in Natural Language Requirements," in *Proceedings of the Thirteenth International Conference on Software and Systems Engineering and Applications*, Paris, France (2000).

[18] Kamsties, E., von Knethen, A., Philipps, J., and Schaetz, B., "An Empirical Investigation of the Defect Detection Capabilities of Requirements Specification Techniques," in *Proceedings of the Sixth CAiSE/IFIP8.1 International Workshop on Evaluation of Modelling Methods in Systems Analysis and Design (EMMSAD'01)*, Interlaken, CH (June 2001).

[19] Kamsties, E., "Surfacing Ambiguity in Natural Language Requirements," Ph.D. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany (2001).

[20] Lyons, J., *Semantics I and II,* Cambridge University Press, Cambridge, UK (1977).

[21] Nakajima, T. and Davis, A.M., "Classifying Requirements Errors for Improved SRS Reviews," pp. 88–100 in *Proceedings of the First International Workshop on Requirements Engineering: Foundation of Software Quality (REFSQ)*, Utrecht, NL (June 1994).

[22] Parnas, D.L. and Weiss, D.M., "Active Design Reviews: Principles and Practices," pp. 132–136 in *Proceedings of the Eighth International Conference on Software Engineering*, London, UK (August 1985).

[23] Parnas, D.L., Asmis, G.J.K., and Madey, J., "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety* **32**(2), pp. 189–198 (April–June 1991).

[24] Poesio, A M., *Semantic Ambiguity and Perceived Ambiguity, Semantic Ambiguity and Underspecification,* Cambridge University Press, Cambridge, UK (1996), No 55 in CSLI Lecture Notes..

[25] Robertson, S. and Robertson, J., *Mastering the Requirements Process,* Addison-Wesley, Harlow, England (1999).

[26] Rupp, C. and Goetz, R., "Linguistic Methods of Requirements-Engineering (NLP)," in *Proceedings of the European Software Process Improvement Conference (EuroSPI)*, Denmark (November 2000).

[27] Schneider, G.M., Martin, J., and Tsai, W.T., "An Experimental Study of Fault Detection in User Requirements Documents," *ACM Transactions on Software Engineering and Methodology* **1**(2), pp. 188–204 (April 1992).

[28] Sommerville, I. and Sawyer, P., *Requirements Engineering: A Good Practice Guide,* John Wiley & Sons (1997).