Chapter 2

AMBIGUITY IN REQUIREMENTS SPECIFICATION

Daniel M. Berry [1] and Erik Kamsties [2]
[1] *School of Computer Science - University of Waterloo - Waterloo, Ontario N2L 3G1, Canada*
[2] *Software Systems Engineering - University of Essen - D-45117 Essen, Germany*

Abstract: This chapter identifies the problem of ambiguity in natural language requirements specifications. After observing that ambiguity in natural language specifications is inescapable when producing computer-based system specifications, a dictionary, linguistic, and software engineering definitions of ambiguity are given. The chapter describes a full range of techniques and tools for avoiding and detecting ambiguity.

1. NATURAL LANGUAGE REQUIREMENTS SPECIFICATIONS

Software requirements specifications (SRSs) need to be precise and accurate, to be self-consistent, and to anticipate all possible contingencies. The overwhelming majority of requirements specifications are written in natural languages, albeit often amplified by information in other notations, such as formulae and diagrams. Only occasionally, one finds a completely formalized SRS using very little natural language, except as commentary. Virtually every conception for a system is written in a natural language. Virtually every request for proposal is written in natural language. A recent on-line survey of businesses requiring software, conducted at Università di Trento in Italy and available at eprints.biblio.unitn.it [53], shows that a majority of documents available for requirements analysis are provided by the user or are obtained by interviews. Moreover,

- 71.8% of these documents are written in common natural language,
- 15.9% of these documents are written in structured natural language, and
- only 5.3% of these documents are written in formalized language.

J. C. S. Prado Leite et al. (eds.), *Perspectives on Software Requirements*

However, we all know that natural language is *so* imprecise, so ambiguous, and so inherently so! Thus, even if the authors of an SRS are trying to avoid ambiguities, unintended ambiguities sneak up behind us in the SRS.

1.1 Ambiguities in Natural Language Requirements Specifications

Unintended ambiguity is the Achille's heel of SRSs, leading to diverging expectations and inadequate or undesirably diverging implementations. Unintended ambiguity admits multiple interpretations of the underlying document. An ambiguous SRS can lead to two or more implementors writing interfacing code to operate under different assumptions, despite each's being confident of having programmed the correct behavior.

The ambiguity problem is exacerbated if, for some reason, the author of the SRS is unavailable for questioning at the time the SRS is being interpreted during implementation. It is quite common for the requirements engineers who wrote an SRS to move on to other projects and to be unavailable for questioning by the implementors.

Don Gause [25] lists the five most important sources of requirements failure as:

- failure to effectively manage conflict,
- lack of clear statement of the design problem to be solved,
- too much unrecognized disambiguation,
- not knowing who is responsible for what, and
- lack of awareness of requirements risk.

It is instructive that unrecognized disambiguation ranks as high as the other, universally recognized sources of failure.

Unrecognized or *unconscious* disambiguation is that process by which a reader, totally oblivious to other meanings of some text that she has read, understands the first meaning that comes to mind and takes it as the only meaning of the text. That unconsciously assumed meaning may be entirely wrong.

There is that old tradeoff: requirements specifications written in natural languages versus mathematics-based formal languages. On one hand, natural language is inherently ambiguous, but there is always someone who can write it, and it is always more or less understood by all stakeholders, albeit somewhat differently by each. On the other hand, mathematics-based formal language is inherently unambiguous, but there is not always someone who can write it, and it is not understood by most stakeholders, although all that do understand it understand it the same. A lot of research in requirements engineering (RE) is directed at solving the problem of imprecise ambiguous requirements specifications by convincing people

to use mathematics-based formal languages and addressing the negatives of formal languages. These issues are outside the scope of this chapter.

However, even if mathematics-based formal languages are used for requirements specifications, the reality is that there is no escaping natural language requirements specifications. Michael Jackson [42] reminds us that "Requirements engineering is where the informal meets the formal." Therefore, natural languages are inevitable, even if it is only for the initial conception. (Unless the client is some really weird math-type nerd who thinks in first-order predicate calculus.) Even if one moves immediately to formal languages, the inherent ambiguity of the natural language initial conception can strike as the transition is made. What the formalizer understands of the conception may be different from what the conceiver meant.

On the basis of this observation, one could argue that the subject for elimination of ambiguities is not just the SRS, but *all* requirements documents, including the preliminary documents ranging from the description of initial concepts to the complete SRS and the companion documents, such as the user's manual. However, we should be neither too inclusive nor too exclusive. It is proper to live with ambiguities in preliminary requirements documents, just as we live with inconsistency in these documents [14], as forcing linguistic precision too early could inhibit creativity. It is equally proper to insist that also SRS companions have no ambiguities. After all, it is not good if the user is left with a misunderstanding of a system from an ambiguous user's manual. Nevertheless, the most critical requirements document to be unambiguous is the SRS. Therefore, this chapter uses the term "SRS" to stand for all requirements documents that need to be unambiguous.

What about semi-formal languages such as the UML models? They give a double whammy. Ambiguity can still strike when we go from the conception to the models. Moreover, because UML is ill-defined and inconsistently used, the models themselves are not unambiguous.

Therefore, there is a group of researchers focusing on solving the problem of imprecise, ambiguous requirements specifications by trying to improve our writing, understanding, and processing of natural languages. There are several approaches to avoiding the ambiguity and imprecision of natural languages:

1. Learn to write less ambiguously and less imprecisely.
2. Learn to detect ambiguity and imprecision.
3. Use a restricted natural language which is inherently unambiguous and more precise.

The first two reduce the disadvantage of existing natural languages. The third restricts the natural languages to disallow the disadvantages. However, by restricting the natural language, it is not so natural anymore. Then the question of expressibility comes up: Have we lost something valuable in the restrictions? How convenient is it for humans to write in the language? Is writing in such a language really different

from writing in a mathematics-based formal language? Thus, the focus of this chapter is on writing less ambiguously and less imprecisely and on detecting ambiguity and imprecision.

1.2 Plan for the Rest of the Chapter

The plan for this chapter is: Section 2 makes precise the assumed background of this chapter. Section 3 explores several definitions of ambiguity, ranging from a dictionary definition, through software engineering definitions, to various linguistic definitions. Section 4 explores a conceptual model of software engineering ambiguities. Section 5 describes several techniques for avoiding and detecting ambiguities in requirements documents. Section 6 concludes with a summary and a discussion of practical implications.

In order to avoid using variants of "he or she" as a third person singular subject pronoun of nondeterminate gender, only "he" or only "she" is used in any one text. The gender of the third person singular alternates by section, starting with "she" in Section 1.

Rather than surrounding all examples and words used as themselves with quotation marks, this report adopts a convention of typesetting examples and some words used as themselves in a sans-serif font, like these four words, while leaving the general narrative in a serif font, like these four words.

2. DEFINITIONS AND BACKGROUND

This section gives the background understanding about SRSs that is assumed of the reader and that motivates much of the concern about ambiguous requirements and many of the decisions made when there are choices to be made.

2.1 Software and System Requirements Specifications

By "software requirements specifications (SRSs)", we mean specifications for mainly the software component of computer-based systems (CBSs). Strictly speaking, such a specification may deal with non-software, system issues. However, it is usual to talk about these specifications as software specifications, mainly because the most changeable part of a CBS is its software. The specifications state what can be assumed of the environment and what must be done by the system, generally in the form of a description of the system's response to any input.

2.2 Sources of Information

SRSs are written by one or more professionals in the software field for a client who is not in the field. Occasionally, a professional in the relevant application domain participates in the writing. SRS writing requires the professional to elicit information from the client's representatives and to observe, interview, and negotiate with them. It requires the client's representatives to validate that the professionals have captured the client's intent.

2.3 Changes

A common and pervasive problem with software specifications is that of change. Requirements always change [41], and keeping up-to-date and consistent the requirements specification and all other documents that must be kept consistent with the requirements specification is a major problem and is the subject of a whole branch of software engineering (SE), *change management* [11]. For software, change seems relentless all through the lifecycle.

2.4 Fixing Errors

In SE, it is known that the cost to fix a detected error goes up exponentially with the lifecycle stage [7]. That is, an error that costs X (where X is any measure of work or resources) to fix during requirements specification, costs about $10 \cdot X$ to fix during coding and about $200 \cdot X$ to fix after deployment. Therefore, it pays handsomely to find errors during requirements specification, so handsomely that it is worth spending a significant fraction of the total budget on requirements specification, e.g., even as much as 90%. Nevertheless, it is rare to spend more than 25% of the total budget on requirements specification because there is a perception that no one on the project is really working until the implementation of frozen requirements begins. The irony here, of course, is that requirements change so relentlessly that "frozen requirements" is an oxymoron.

3. DEFINITIONS OF AMBIGUITY

Ambiguity is a real-world phenomenon that rears its ugly head in many disciplines including writing, linguistics, philosophy, law, and—of course—software engineering. Ambiguity is a multi-faceted problem. In order to learn to write unambiguously or to detect ambiguity, a detailed understanding of the various facets of ambiguity is required. This section provides a comprehensive review of ambiguity definitions from different sources and research disciplines, namely from an English dictionary, from linguistics, and from software engineering.

We limit the discussion to natural language text, although it is clear that semi-formal and even formal descriptions are open to ambiguities, albeit a more limited number.

Note that the term "ambiguity" should probably be "multiguity" to remove the implication, arising from the prefix "ambi", that there are only two meanings. However, we stick to the conventional word with the understanding that we are really talking about multiguities.

3.1 Dictionary Definition

The Merriam Webster English Dictionary [1] defines "ambiguity" as

1a. the quality or state of being ambiguous especially in meaning,
1b. an ambiguous word or expression, or
2. uncertainty,

where "ambiguous" means

1a. doubtful or uncertain especially from obscurity or indistinctness <eyes of an ambiguous color>,
1b. inexplicable, or
2. capable of being understood in two or more possible senses or ways.

Thus, there are two basic interpretations of "ambiguity",

1. the capability of being understood in two or more possible senses or ways and
2. uncertainty.

Uncertainty in the sense of lack of sureness about something is ignored here, because this lack of sureness has to do with the writer's and reader's knowledge about the background; the requirement sentence itself could be precise and clear.

3.2 Linguistic Definitions

Linguistic ambiguity is investigated in linguistics [51] and in related fields, namely, computational linguistics [35,2] and philosophy [50,69]. Space limitations prevents us from reviewing ambiguity in all of these fields. Therefore, first, the types of ambiguity considered in linguistics are discussed. Second, related phenomena, such as vagueness and some observations by the authors, are described.

One can distinguish four broad classes of linguistic ambiguity, lexical ambiguity, syntactic ambiguity, semantic ambiguity, and pragmatic ambiguity. Note that this classification is not mutually exclusive. Rather, the ambiguity or ambiguities occurring in a single ambiguous text may be a combination of several kinds.

3.2.1 Lexical Ambiguity

Lexical ambiguity occurs when a word has several meanings. Lexical ambiguity can be subdivided into homonymy and polysemy.

1. *Homonymy* occurs when two different words have the same written and phonetic representation, but unrelated meanings and different etymologies, i.e., different histories of development. For example, the etymology of bank in the sense of an establishment for custody, loan, exchange, or issue of money is, according to Merriam-Webster's dictionary [1],

   > Middle English, from Middle French or Old Italian; Middle French *banque*, from old Italian *banca*, literally bench, of Germanic origin; akin to Old English *benc*

   while the etymology of bank in the sense of a rising ground bordering a lake, river, or sea is

   > Middle English, probably of scandinavian origin; akin to Old Norse *bakki* bank; akin to Old English *benc* bench—more at BENCH Date: 13th century

2. *Polysemy* occurs when a word has several related meanings but one etymology. For example, green has several different, but related, meanings with a common etymology, according to Merriam-Webster's dictionary, such as

   1. of the color green
   2. pleasantly alluring
   3. youthful, vigorous
   4. not ripened or matured

   Systematic polysemy occurs when the reason for the polysemy is confusion between classes, e.g., between unit and type, and between process and product [10]. As an example of unit-vs.-type ambiguity, I like this jacket can refer to an individual jacket or to a type of jacket. Process-vs.-product ambiguity occurs with everyday words like building, shot, and writing. The noun form of any of these can refer to a process or to the product of the same process. Another systematic polysemy is the behavior-vs.-disposition ambiguity. For example, This is a fast car can denote the current behavior of a particular car or the general capability of each element of a class of cars, such as Ferrari, independent of the current behavior of any particular Ferrari.

3.2.2 Syntactic Ambiguity

Syntactic ambiguity, also called *structural ambiguity*, occurs when a given sequence of words can be given more than one grammatical structure, and each has a

different meaning. In the terminology of compiler construction, syntactic ambiguity occurs when a sentence has more than one parse. Syntactic ambiguity includes attachment ambiguity and coordination ambiguity, among other kinds.

1. *Attachment ambiguity* occurs when a particular syntactic constituent of a sentence, such as a prepositional phrase or a relative clause, can be legally attached to two parts of a sentence. The most popular pattern of attachment ambiguity is a prepositional phrase that may modify either a verb or a noun. For instance, in the sentence

   (1) The police shot the rioters with guns.

   the phrase with guns can be taken either as a modifier of the noun rioters or as a modifier of the verb shot, leading to two different interpretations: either the rioters were armed with guns when the police shot them, or the police used guns to shoot the rioters. More patterns are described by Hirst [35].
2. *Coordination ambiguity* occurs
   a) when more than one conjunction, and or or, is used in a sentence or
   b) when one conjunction is used with a modifier.
   An instance of the first kind is the sentence:

   (2) I saw Peter and Paul and Mary saw me.

   This sentence can be read either as I saw (Peter and Paul) and Mary saw me or as I saw Peter and (Paul and Mary) saw me. A well-placed comma disambiguates a sentence like this one.
   An instance of the second kind is the phrase:

   (3) young man and woman

   It can be read either as (young man) and woman or as young (man and woman).

3.2.3 Semantic Ambiguity

Semantic ambiguity occurs when a sentence has more than one way of reading it within its context although it contains no lexical or structural ambiguity. Semantic ambiguity can be viewed as ambiguity with respect to the logical form, usually expressed in predicate logic, of a sentence. Semantic ambiguity can be caused by

1. coordination ambiguity,
2. referential ambiguity, and
3. scope ambiguity.

Coordination ambiguity is already discussed. Referential ambiguity is on the borderline between semantic and pragmatic ambiguity, because referential

ambiguity can happen within a sentence or between a sentence and its discourse context. Thus, referential ambiguity is discussed in the section on pragmatic ambiguity.

The *quantifier operators* include such words as every, each, all, some, several, a, etc., and the *negation operators* include not. *Scope ambiguity* occurs when these operators can enter into different scoping relations with other sentence constituents. For example, in the sentence

(4) All linguists prefer a theory.

the quantifiers all and a interact in two ways. When the scope of a includes the scope of all, this sentence means all linguists love the same one theory. When the scope of all includes the scope of a, this sentence means that each linguist loves a, perhaps different, theory.

3.2.4 Pragmatic Ambiguity

Pragmatics is the study of the relations between language and context [50]. The research area of pragmatics has linguistic and philosophical roots. The scope of pragmatics, i.e., the phenomena that are considered pragmatic, and its boundary with semantics are subjects of ongoing discussions. In general, both pragmatics and semantics investigate the meaning of language; pragmatics is concerned with context-dependent meaning, while semantics is concerned with context-invariant meaning. With definitions such as these, it is clear that some phenomena, not discussed here for the lack of space, straddle the semantics–pragmatics boundary.

Pragmatic ambiguity occurs when a sentence has several meanings in the context in which it is uttered. The context comprises the language context, i.e., the sentences uttered before and after, and the context beyond language, i.e., the situation, the background knowledge, and expectations of the speaker or hearer and the writer or reader. There are more topics of pragmatics, such as conversational implicature, speech acts, and conversational structure, which we do not discuss, because they do not concern requirements documents.

In traditional semantics, the relation between a word or phrase and the object of the real world that the word or phrase describes is called a *reference*. An *anaphor* is an element of a sentence that depends for its reference on the reference of another element, possibly of a different sentence. This other element is called the *antecedent* and must appear earlier in the same sentence or in a previous sentence. A *referential ambiguity* occurs when an anaphor can take its reference from more than one element, each playing the role of the antecedent. Anaphora include pronouns, e.g., it, they, definite noun phrases, and some forms of ellipses. An example of a referential ambiguity is:

(5) The trucks shall treat the roads before **they** freeze.

The antecedent of the anaphor they can be either trucks or roads. Anaphora can refer also to sets of objects, compound objects, or verbs.

Ellipses can have the same effect as pronouns and definite nouns, as the following sentence shows.

(6) ... 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 that.

The following example has elements of scope and referential ambiguities, among others.

(7) Every student thinks she is a genius.

When the scope of the quantifier every includes she, she refers, as an anaphoric expression, to student, which is thus known to be feminine. When the scope of the quantifier every does not include she, she refers to a contextually specified specific female, who may or may not be one of the students.

3.2.5 Vagueness and Generality

Vagueness and *generality*, also called *indeterminacy*, are closely related to ambiguity. Cruse distinguishes a general word or phrase from an ambiguous word or phrase [10]. Each is open to more than one interpretation. However, they have different statuses in human communication as shown by cousin and bank in the following sentences:

(8) Sue is visiting her cousin.

(9) We finally reached the bank.

Cousin, in the first example, can refer to either a male or a female cousin. However, the sentence can function as a satisfactory communication without either the hearer perceiving, or the speaker intending to convey, anything concerning the gender of the referent, because cousin has a general meaning. The general meaning covers all the specific possibilities, not only with regard to gender, but also with regard to an unbounded number of other matters, such as height, age, eye color, etc.

Also bank, in the second example, can be interpreted in more than one way, as we have seen before, as a river bank or as a money bank, but it has no general meaning covering these possibilities. Furthermore, the interpretation cannot be left undecided. Each of the speaker and hearer must select a meaning. Moreover, they must select the same reading if the sentence is to play its role in a normal conversational exchange.

Cousin is *general* with respect to gender. Bank is *ambiguous*. In other words, the two meanings, male cousin and female cousin, are associated with the same word cousin, whose meaning is more general than that of either; they do not represent distinct senses of cousin. On the other hand, the meanings of bank are distinct.

A statement is considered *vague* if it admits *borderline cases* [3]. For example, tall is vague, because a man of 1.8 meters in height is neither clearly tall nor clearly not tall. No amount of conceptual analysis and empirical investigation can settle whether a 1.8-meter man is tall. Moreover, tallness is culture relative. A person regarded as tall in a pygmy tribe would be considered short in an NBA basketball team. Borderline cases are inquiry resistant, and the inquiry resistance recurses. That is, in addition to the lack of clarity of the borderline cases, there is lack of clarity as to where the lack of clarity begins [67].

Linguistic vagueness has a special application in software requirements. A requirement is vague if it is not clear how to measure whether the requirement is fulfilled or not. A non-functional requirement, e.g., fast response time, is often vague, because there is no precise way of describing and measuring it, short of an arbitrary quantification, which leaves us wondering if the essence of the requirements have been captured.

The only difference between vagueness and generality is the existence of borderline cases. A vague expression is inquiry resistant because of borderline cases, while a general expression can be made more precise. For example, cousin is general, but can be described more precisely when necessary. Certainly, there is no doubt as to whether or not a person is a cousin. Moreover, there is no doubt as to which cousins are male and which are female. That is, cousin is not vague.

3.2.6 Language Error

Our experience has identified another category of ambiguity, *language error*. As is the case with the categories described in Section 3.2, language error may not be mutually exclusive of other categories. A language error ambiguity occurs when a grammatical, punctuation, word choice, or other mistake in using the language of discourse leads to text that is interpreted by a receiver as having a meaning other than that intended by the sender.

We have taken the liberty of classifying a language error ambiguity as a pragmatic ambiguity because, like a referential ambiguity, the presence of ambiguity depends on the context; in some contexts, a given text may have a language error, and in another context, the same text may not have a language error. We discuss it in the section about linguistic ambiguity, as opposed to in the section on software engineering ambiguity, because these errors occur in all kinds of documents and not just software engineering documents.

For example,

(10) Every light has their switch.

has a grammatical error that is commonly committed by present-day, even native, English speakers. The error is that of considering every *X*, which is singular, as plural although it precedes a correct singular verb, as in

(11) Everybody brings their lunch.

In the case of

Every light has their switch. ,

the reader does not know if the intended meaning is

(12) Every light has its switch. ,

which is the same as

(13) Each light has its switch. ,

or

(14) All lights have their switch. ,

which could mean either:

(15) All lights share their switch.

or

(16) Each light has its own switch.

Basically, because of the error, the reader does not know how many switches there are per light.

Many times, a language error ambiguity is at the same time another kind of ambiguity. The sender does not know an error has been committed, and the receiver may or may not know that an error has been committed. If the receiver does not know, she may or may not understand it as intended. If she does know, she may or may not be able to make a good guess as to what is intended, but in the end she may be left wondering.

The reason this new category is needed is that sometimes there is a language error and no other kind of ambiguity. Sometimes, there is a linguistic mistake only if the intention is one way but not if it is another way. For example, in

(17) Everybody brings their lunch. ,

everyone knows that the intended meaning is

(18) Everybody brings her lunch. ,

even though their, being plural, is incorrectly used with the singular Everybody; here we have a language error without an ambiguity. In

(19) I only smoke Winstons.

if the intention is to say,

(20) I smoke only Winstons.

there is the language error of a misplaced only. However, if the intention is to make it clear, in an admittedly strange conversation about eating Winston cigarettes, that one only smokes and does not eat Winstons, then there is no language error. However, someone not privy to the whole conversation, and hearing only

I only smoke Winstons.

may understand

I smoke only Winstons.

which would be contrary to the intention, even though the intention is in fact what is said by the sentence, according to the rules about placement of only.

3.3 Software Engineering Definitions

There appears to be no single comprehensive definition of ambiguity in the software engineering literature. Each of the following definitions highlights only some aspects of ambiguity and omits others. The definitions together form a complete overview of the current understanding of ambiguity in SE. The key natural language document whose ambiguity can be a show stopper in software engineering is the requirements specification. Therefore, the software engineering definition both has relevance to requirements engineering and draws from requirements engineering concerns, of making unambiguous specifications of well-understood requirements.

3.3.1 IEEE Definition

The IEEE Recommended Practice for Software Requirements Specifications [37] says that "An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation." Presumably, an SRS is ambiguous if it is not unambiguous.

The problem with the IEEE definition is that there is no unambiguous specification simply because for any specification, there is always someone who understands it differently from someone else. Parnas says that he has never seen a bug-free program [59]. There *are* mature, usable programs whose bugs are known; the users have learned to work around the bugs and get useful computation from them. In a similar manner, there are no unambiguous specifications. Rather, there

are *useful* specifications. A useful specification is a specification that is understood well enough by enough people that count, a majority of the implementors, a majority of the customers, and a majority of the users, that it is possible to implement software meeting the specifications that does what most people expect it to do in most circumstances.

3.3.2 Davis's Definition

Indeed, Davis [12] has suggested a test for ambiguity to serve as a definition: "Imagine a sentence that is extracted from an SRS, given to ten people who are asked for an interpretation. If there is more than one interpretation, then that sentence is probably ambiguous." The problem with this test is that, as in software testing, there is no guarantee that the eleventh person will not find another interpretation. However, this test does capture the essence of a useful SRS, which is unambiguous for most practical purposes. Davis provides two examples of ambiguity.

(21) For up to 12 aircraft, the small display format shall be used. Otherwise, the large display format shall be used.

Assuming that small and large display formats are defined previously, the ambiguity lies in the phrase for up to 12. Does it mean for up to and including 12 or for up to and excluding 12?

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

Assuming again that the relevant terms are defined, the ambiguity lies in the relative precedence of and and or, because we cannot assume the precedence rules of Boolean algebra for natural language utterances. This second example is used frequently in the rest of this chapter.

3.3.3 Schneider, Martin, and Tsai's Definition

Schneider, Martin, and Tsai [65] give another definition of ambiguity. "An important term, phrase, or sentence essential to an understanding of system behavior has either been left undefined or defined in a way that can cause confusion and misunderstanding. Note, these are not merely language ambiguities such as an uncertain pronoun reference, but ambiguities about the actual system and its behavior."

In addition to the IEEE definition, Schneider, Martin, and Tsai identify two possible categories of ambiguity, namely language ambiguities and software engineering ambiguities. Language ambiguities can be spotted by any reader who has an ear for language. In contrast, software engineering ambiguities depend on the

domain involved and can be spotted only by a reader who has sufficient domain knowledge. Parnas, Asmis, and Madey give an example of a software engineering ambiguity. Their example is a requirement that was introduced without the reader's having been told that the water level that is the subject of the requirement varies continuously [58].

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

This sentence has at least four interpretations.

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.

The software engineers building the pump 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 under this interpretation. 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. We, the authors of this chapter, actually did not see any ambiguity at all and assumed the fourth interpretation. We did not know that there were waves in the tank, because the text leading up to the example made no mention of waves. If the water were quiescent except when water is being added and its top surface were always essentially flat, all four interpretations would be identical in meaning, and the fourth interpretation provides the easiest implementation. Clearly, domain knowledge is needed to know that there are waves and thus, that the third interpretation is intended.

3.3.4 Gause and Weinberg's Definition

According to Gause and Weinberg, ambiguity has two sources, *missing information* and *communication errors* [24]. Missing information has various reasons. For instance, humans make errors in observation and recall, tend to leave out self-evident and other facts, and generalize incorrectly. A communication error that occurs between the author and the reader is due to general problems in the writing.

Gause and Weinberg [24] give an example of an ambiguity due to missing information.

> (24) Create a means for protecting a small group of human beings from the hostile elements of their environment.

They provide three interpretations:

1. an igloo (an indigenous home constructed of local building materials),
2. a Bavarian castle (a home constructed to impress the neighbors), and
3. a space station (a mobile home with a view).

The reason for these wide-ranging interpretations of the requirement is that a lot of issues are missing, namely the cost, material, size, shape, and weight of the means of protection, the other functions that shall be performed inside this means, and the nature of the environment. Moreover, the phrase small group is an example of an expression inadequacy; are we talking about 5, 20, or 100 people?

The view of Gause and Weinberg is shared by Harwell, Aslaksen, Hooks, Mengot, and Ptack [33] in their definition of an unambiguous requirement. "A requirement must be unambiguous in the sense that different users (with similar backgrounds) would give the same interpretation to the requirement. This definition has two aspects. On one hand, there is the aspect of grammatical ambiguousness, i.e. the poorly constructed sentence. On the other hand, there is the aspect of ambiguousness arising from a lack of detail allowing different interpretations. The first of these can be measured or tested independently of author or user, but the second aspect can be measured only in conjunction with a set of users, since it depends on what assumptions the user makes automatically, i.e. as a result of the user's background knowledge."

3.3.5 Kamsties's Definition

Kamsties defines a requirement as ambiguous if it has multiple interpretations despite the reader's knowledge of the context [46]. It does not matter whether the author unintentionally introduced the ambiguity, but knows what was meant, or she intentionally introduced the ambiguity to include all possible interpretations. The context is important to be taken into account, because a requirements document cannot be expected to be self-contained in a way that an arbitrary naïve reader could understand it.

Based on this definition, Kamsties proposed a comprehensive taxonomy, shown in Figure 2-1, of types of ambiguities that may appear in requirements [46]. This taxonomy includes the linguistic effects that can make a requirements statement ambiguous, and it includes a classification of the software engineering ambiguities. Software engineering ambiguities are classified by the context that must be considered when considering a requirements statement. These contexts are

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

This understanding of context is inspired by the WRSPM (World, Requirements, Specifications, Program, and Machine) model [31] and by the Four-World model [43].

If a requirement statement has several interpretations even though the reader has all necessary contextual knowledge, then it is ambiguous. The software engineering ambiguity that an ambiguous requirements statement has is named for the context whose contextual knowledge is required to interpret the requirements statement.

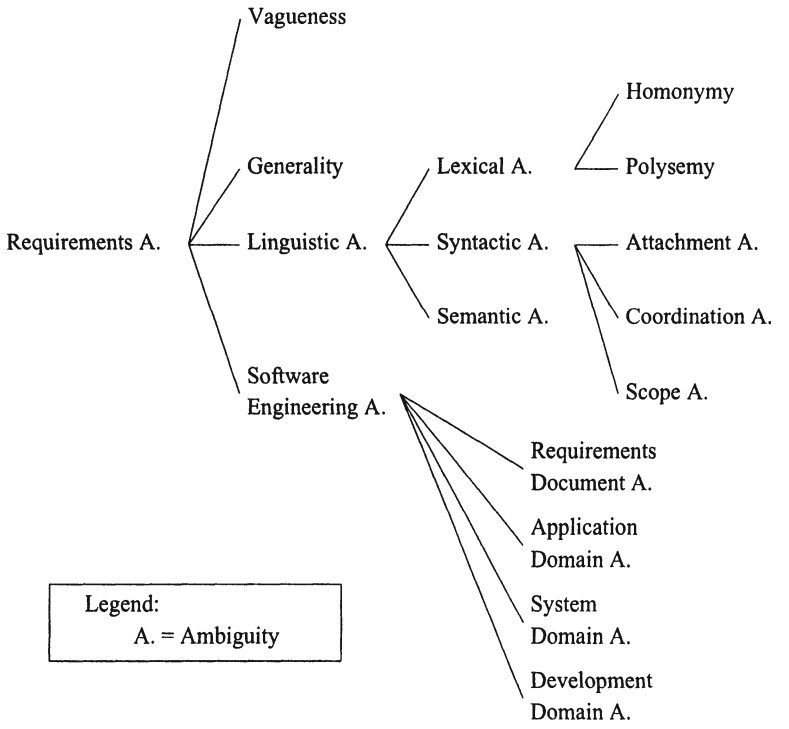

Figure 2-1. Taxonomy of Ambiguity Types

Normally, also real-world knowledge and language knowledge help to disambiguate an ambiguity. However, it is assumed that this knowledge is sufficiently shared by requirements authors and readers. Therefore, real-world and language knowledge are ignored in this definition.

A single requirement is rarely self-contained. Usually, it has implicit or explicit references to other requirements. That is, the reader must know the related requirements in order to understand a requirement correctly.

A *requirements document ambiguity* occurs if a requirement allows several interpretations with respect to what is known about other requirements in the requirements document. Requirements document ambiguity can arise from pronoun references, e.g., it, and definite noun phrases like the one below. The requirement

(25) the product shall show all roads predicted to freeze.

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

An *application domain ambiguity* occurs if a requirement allows several interpretations with respect to what is known about the application domain. The requirement of Example 23 contains an example of such an ambiguity. As discussed above, it is observable only to a person who has application domain knowledge.

A *system domain ambiguity* occurs if a requirement allows several interpretations with respect to what is known about the system domain. The requirement

(26) If the timer expires before receipt of a disconnect indication, the SPM requests transport disconnection with a disconnect request. The timer is cancelled on receipt of a disconnect indication.

is ambiguous. The ambiguity arises from the system domain. It is ambiguous whether or not the second sentence is part of the if-statement in the first sentence. This particular requirement could be disambiguated by the application of common sense; the cancellation of an expired timer probably makes little sense, but the sentence illustrates the issue nicely.

A *development domain ambiguity* occurs if a requirement allows several interpretations with respect to what is known about the development domain.

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

It remains 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. That is, the statement can be interpreted as either indicative or optative [39]. 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 will be true about the environment in the future.

3.3.6 The Role of Ambiguity in Software Engineering

A glance through the requirements engineering literature shows that different interpretations of ambiguity are in actual use. Some, including Davis [12] and Parnas, Asmis, and Madey [58] argue that ambiguity is not acceptable at all because of the disastrous consequences that misinterpretations can have in software design and implementation. Here, the term "ambiguity" is interpreted as ambiguity caused by expression inadequacy, as in Example 24

Others, including Goguen [28] and Mullery [56], and Gause [25] argue that ambiguity is acceptable for a while, as are inconsistencies [14,15]. Here, the term "ambiguity" is interpreted as ambiguity caused by missing information, as in Example 24. Goguen points out that "although natural language is often criticized, e.g. by advocates of formal methods, for its informality, ambiguity and lack of explicit structure, the features can be advantages for requirements. For example, these features can facilitate the gradual evolution of requirements, without forcing too early a resolution of conflicts and ambiguities that may arise from the initial situation; it is important not to prejudge the many trade-offs that will have to be explored later, such as cost versus almost everything else.... And finally, natural language can permit the 'diplomatic' resolution of conflicts through the careful construction of deliberate ambiguities; for example this is rather common in large government financed projects."

Mullery points out that a certain degree of ambiguity, and incompleteness and inconsistency as well, must be accepted over a period of time, which may well extend through into design and implementation stages. An unambiguous SRS is unattainable, because requirements are constantly changing. The quest for an unambiguous SRS, what is usually desired for contractual software development, costs additional effort. However, the total effort for producing a SRS is defined in the delivery contract; penalties for failure to deliver the SRS in time are specified. Thus, the additional effort to achieve a lack of ambiguity must be estimated carefully, keeping in mind the inevitable changes, both to the domain and to the system functionality. Attempting to achieve an unambiguous SRS results in far more work whenever a change is needed, or even worse, it makes keeping abreast of needed changes totally infeasible. Don Gause says that conflict is good because it exposes problems with the requirements [25]. On the other hand, too much unconscious ambiguity leads to failure, because people make unconscious design decisions or make conscious design decisions based on unconscious design assumptions, all based on unconscious disambiguation based on their initial reading.

In summary, there are two major types of ambiguities, language ambiguities and software engineering ambiguities. Some authors consider only expression

inadequacy as a source of ambiguity, others consider missing information as an additional source. That is, "ambiguity" can mean two slightly different things; ambiguity is ambiguous!

4. EXPLORING SOFTWARE ENGINEERING AMBIGUITIES

In the requirements documents that the authors have investigated, software engineering ambiguities accounted for a majority of the ambiguities, while pure linguistic ambiguities played a less significant role. Moreover, these ambiguities are context-specific. That is, requirements for an information system may contain types of ambiguities different from those of requirements for an embedded system. This section reviews an approach for identifying the software engineering ambiguities that are specific to a particular context. As mentioned in Section 3, the more detailed is the understanding of possible ambiguities, the more likely the ambiguities can be avoided or detected.

First, we discuss the nature of software engineering ambiguities more deeply using an analogy to machine translation. The following sentence contains two ambiguities.

(28) John hit the wall.

The first one is a linguistic ambiguity, in particular, a lexical one that concerns hit. The word is ambiguous, because it has transitive senses, e.g., to reach, and intransitive senses, e.g., to attack or to strike. The second ambiguity arises for a German reader from wall. The word is translated into German as Wand if the wall is inside a building and it is translated as Mauer if the wall is outside. This type of ambiguity is a not a linguistic one. In machine translation, it is called a *conceptual translational* ambiguity. It is, by no means, a rare phenomenon, even between closely related languages [36]. The same situation occurs with requirements documents. The customers and users speak a language different from that spoken by the requirements engineers and developers. As in Example 28, ambiguities can be already contained in the requirements as uttered by the customers or they can occur due to the translation into the requirements engineers' language. The translational ambiguities that occur due to the translation from a requirement in the customer's language to a requirement in a requirements engineer's language are called software engineering ambiguities. Example 22, given in Section 3.3, and repeated here to avoid page flipping on the reader's part, contains at least two ambiguities,

(22) 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, three of which are shown in Figure 2-2.

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 hand and middle state diagrams, or precedes and, but in the right hand state diagram, and precedes or. Second, there is a software engineering ambiguity that concerns the alert. In some contexts, it is important to distinguish actions from activities. A short, non-interruptible action is executed when a transition fires, a state is entered, or a state is exited. A longer, interruptible activity is performed while being in a state. The phrase raise an alert can be interpreted as an action on the transition to the next state, as in the left hand state diagram, as an action upon entry to the next state, as in the middle state diagram, or as an activity in the next state, as in the right hand state diagram.

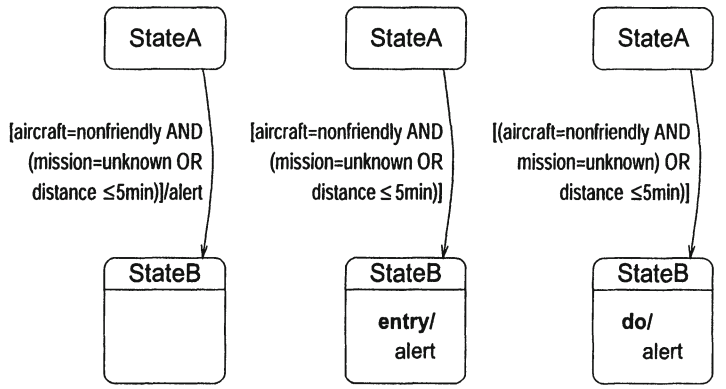

Figure 2-2. UML State Diagrams for Example 22

Kamsties has developed an approach for identifying types of software engineering ambiguities that may occur in requirements documents of a particular context [46]. This approach makes use of the observation that each resolution of an ambiguity in a natural language requirement leads to a different model. The idea is to identify and describe types of ambiguity that may occur in a requirements document from available metamodels that describe the structure of the used requirements document, the particular application domain, the system domain, and the development domain. For example, the UML metamodel contains a lot of knowledge about the system domain. Generally, a metamodel defines a language for specifying a model. As discussed above, 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}, ..., m_{in} in the requirements engineers' language as shown in Figure 2-3.

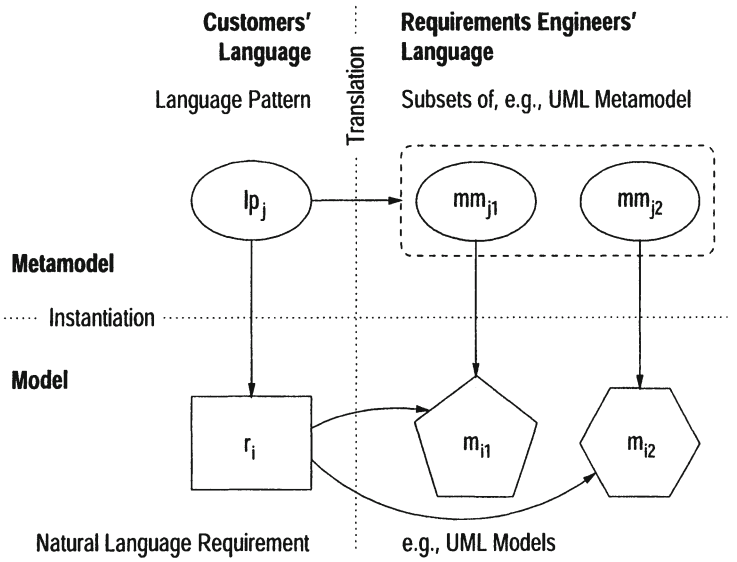

Figure 2-3. *Role of the Metamodel*

From the viewpoint of a metamodel *mm*, the models differ in the submodels, mm_{j1}, mm_{j2}, ..., 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. Figure 2-4 provides an example of a metamodel. This figure depicts a part of a UML metamodel for statechart diagrams. 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 particular concept types and relationships are depicted, which are used in the requirements models of Example 22 shown in Figure 2-2. The three models depicted in Figure 2-2 make use of the same concept types, i.e., state, transition, and action, but their relationships differ. In the left hand model, the action plays the role of an effect of a transition. In the middle model, the action plays the role of an effect of an entry action of a state. In the right hand model, the action plays the role of a doActivity of a state.

The investigation of a metamodel helps to identify types of ambiguity in order to improve the understanding of those types of ambiguities that may occur in a requirements document. 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, can now be made more explicitly.

Kamsties has developed a set of heuristics to help a requirements engineer to systematically identify submodels mm_{j1}, mm_{j2}, ..., 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 remains is the search for ambiguous language patterns lp_j that can be translated into more than one of the identified submodels. Already known patterns should be investigated. Moreover, new patterns emerge as a result of 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.

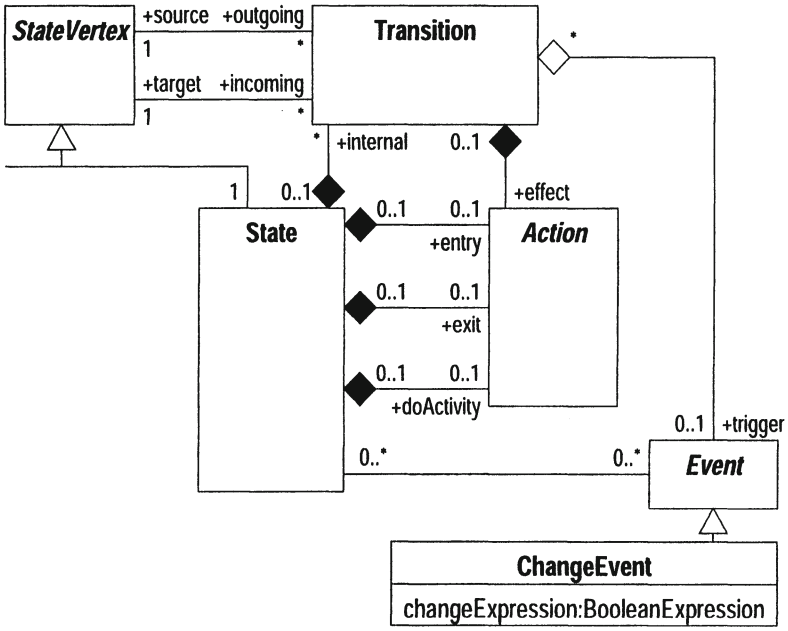

Figure 2-4. Excerpt of the Metamodel of UML Statemachine Package

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 requirements engineering need to be investigated. The more specific the metamodel, the more specific the types of ambiguity that result from the application of the heuristics.

Space limitations prevent us from presenting all heuristics. For more details, see Kamsties's Ph.D. dissertation [46]. In the following, we present the heuristic *Analyze Relationships*, which leads to the identification of the type of ambiguity that causes the ambiguity of the phrase raise an alert in Example 22.

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 2-4 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.

The heuristic can be applied three times on the part of the state machine package depicted in Figure 2-4. 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 exited,
2. during a transition from one state to another,
3. when a state is entered, or
4. while being in a state.

An example of an action ambiguity is Example 22. The identification of the language pattern *verb that describes an action of a system* is the essentially manual task of applying the heuristic.

This approach for identifying types of software engineering ambiguities that may occur in requirements documents was validated on two parts of the UML metamodel, the foundation package and the statemachine package. Moreover, it was applied to the metamodel of SCR, which we extracted from [34]. Seven types of ambiguity were identified in the investigated subset of the UML Foundation package and eleven were identified in the SCR metamodel. The complexities of these metamodels were such that investigating one metamodel required from one to three days.

5. TECHNIQUES FOR DEALING WITH AMBIGUITY

This section presents techniques discovered in related research for reducing the level of ambiguity in natural language requirements, mainly by writing less ambiguously and detecting ambiguities. Techniques can be divided into three groups according to the requirements engineering activities in which they are applied, namely requirements elicitation, requirements documentation, and requirements validation. Sometimes, a strategy can be used in more than one of these activities.

1. For *requirements elicitation*, at least two strategies can be distinguished to minimize ambiguity.
   - First, a context must be established, because language is interpreted always in context, and if this context is not made explicit and agreed to by all the stakeholders in an elicitation session, misinterpretations are likely.
   - Second, the requirements engineer's paraphrasing what she understood from the customers' and users' statements in her own words is effective way for the requirements engineer to get the customers and users to spot their own ambiguities. This technique is usable also for requirements validation. Several communication techniques support this strategy [6,8].
2. For *requirements documentation*, at least three strategies can be distinguished to avoid and detect ambiguity in the written requirements specifications.
   - First, as described in Section 5.1, the precision of natural language can be increased.
   - Second, as described in Section 5.2, more contextual information can be provided in order to allow the reader to resolve ambiguities herself.
   - Third, as described in Section 5.3, conventions on how ambiguous phrases shall be interpreted can be set up between the writers and the readers.
3. For *requirements validation*, at least four strategies can be distinguished to detect ambiguities.
   - First is formalizing informal requirements [66,71]. Since a formal language enforces precision, the hope is that ambiguities can be exposed in the process of being as precise as is necessary for formalization. Indeed, formalization of informal requirements is often reported as being useful to spot defects in the informal requirements [16,71]. However, a recent study [45] shows that this observation is basically true only for certain types of incompleteness and inconsistencies. True ambiguities tend to be spotted less frequently. An ambiguous informal requirement is often not recognized as such, and it ends up becoming an unambiguously wrong formal requirement, if it is unconsciously misinterpreted. That is, the strategy of formalizing requirements does not really help avoid really difficult ambiguities.
   - Second is searching for particular patterns of ambiguity, as done in reading techniques for requirements inspections [24,4,44,46]. As described in Section 5.4, some natural language processing tools can be used to help

search for potential ambiguities. Moreover, the first and second strategies can be combined. Kamsties provides a process for developing formal SCR [34] specifications from informal requirements that helps to spot ambiguities during the formalization [46].

- Third is comparing the interpretations of a document by different stakeholders; if they differ, there is an ambiguity in the original document [24]. Easterbrook and Callahan have combined this strategy with formalization [16]. They applied the third strategy using SCR as the language for expressing an interpretation for validating part of the requirements for the International Space Station [16].
- Fourth is communicating an interpretation back to the requirements author, after which she can easily point out misinterpretations. Also, this strategy can be combined with formalization. Fuchs *et al* developed a formal language that looks similar to natural language, i.e., a controlled natural language, to which natural language specifications can be mechanically formalized [21]. A tool does this translation and resolves ambiguities by a default interpretation. The result is presented to the user to check that the correct interpretation has been chosen.

These last two strategies are perhaps the most effective strategies for finding ambiguities in requirements specifications, but they demand more resources than other strategies and are therefore applicable only in situations in which the cost of system failure as a result of not finding ambiguities is at least the cost of the resources required to find the ambiguities.

Due to space limitations, other than as indicated above, none of these topics is discussed in detail in this chapter. We have chosen to dwell, in the rest of this section, on the most systematic of the techniques.

To illustrate the various strategies for minimizing ambiguities in requirements documentation, a slight variant of Example 22 is used.[1]

(29) An aircraft that is non-friendly and has an unknown mission or the potential to enter restricted airspace within 5 minutes shall raise an alert.

[1] The difference is that Example 22 is written as a plural sentence about all aircraft that meet the conditions, and Example 29 is written as a singular sentence about each aircraft that meets the conditions. A plural sentence has the potential for ambiguity because the correspondence between each element of the universally quantified set and the object or predicate of the sentence is not clear. Does each aircraft raise its own alert or is there a common alert that all aircraft share and raise? While, in this case the intent is clear, there are cases in which the intent might not be clear, in which all members of the set do share something in common, and the reader would be left wondering whether the shared thing or each's own thing is meant. A singular sentence allows describing the exact correspondence and avoids this problem. Note that it would not be good practice for us not to remove ambiguities in our examples!

Even in this form, it suffers from a semantic ambiguity regarding the precedences of and and or.

5.1 Documentation: Increasing the Precision of Natural Language

Glossaries, style guides, sentence patterns, and controlled languages increase the precision and decrease the ambiguity of natural language.

1. A *glossary* or dictionary defines important terms and phrases used in a requirements document. Thus, it helps to avoid lexical ambiguity. It requires considerable effort to create and validate a glossary, but the effort pays off since it can be reused for future projects within the same application domain. For example, Kovitz gives detailed guidelines for creating a glossary [48]. Designations by Jackson [40] and the Language Extended Lexicon by Leite [49,32,47] are other techniques for grounding terms in reality.
2. There are numerous *writing guides*, e.g., *Elements of Style* by William Strunk, Jr. and E.B. White [68], that help authors improve their style and avoid common problems. Some, e.g., *A Dictionary of Modern English Usage*, by H.W. Fowler: [19] deal with ambiguity as a topic. The second edition of Fowler's book, which is already in its third edition, is a favorite of many, and it even has a section on ambiguity as a phenomenon that many consider a gem. Some style guides have been developed specifically to help requirements authors in writing requirements [9,57,48,64]. They can be used also during requirements analysis and validation to check the requirements for possible problems. Finally, some guides have checklists specifically for finding ambiguities [20,30].
   An example of a rule that is found in many style guides is "Use active voice rather than passive voice.", because passive voice blurs the actor in a requirement. A rule that solves the ambiguity problem in Example 29 is "Insert braces to avoid syntactic and semantic ambiguity." Applying this rule to the example yields

   (30) An aircraft that is non-friendly and (has an unknown mission or the potential to enter restricted airspace within 5 minutes) shall raise an alert.

   which makes clear that or binds stronger than and.
3. *Sentence patterns* have been proposed to give the requirements author support in articulating requirements by Rolland and Proix [62] and Rupp and Goetz [63]. An example of a pattern, written in extended Backus-Naur form, for activities that are performed by a system without user interaction, as in the running example, is

(31) [when?] [under what conditions?] THE SYSTEM SHALL | SHOULD | WILL <process> <thing to be processed> [<process detail>*]

The requirement must be rewritten slightly to fit the pattern:

(32) If [an aircraft is non-friendly] and [has an unknown mission or the potential to enter restricted airspace within 5 minutes], the system shall <raise> <an alert>.

Notice that the and–or ambiguity is resolved in the same way as suggested by the rule to use braces.

The set of language patterns of Rolland and Proix was designed for the information systems domain. Denger proposed a set of empirically developed language patterns that is geared specifically towards the embedded control systems domain [13]. The special characteristic of the work is that the set of patterns was not derived from only the author's experience. Rather, it was systematically derived from a metamodel describing the contents of requirements in that domain. If one finds the patterns too limited, the metamodel can be extended and some guidance is provided for updating the set of patterns. Moreover, the completeness of the patterns with respect to the metamodel can be demonstrated.

4. Another approach to increase the precision of natural language is to use a *controlled language*, which is a precisely defined subset of natural language for use in specific environments. The objective of a controlled language is to increase the readability and understandability of any kind of technical documentation. This improvement is accomplished by reducing the inherent ambiguity of natural language through a restricted grammar and a fixed vocabulary. There is a genuine need for tool support for writing requirements documents in order to enforce the grammar and the fixed vocabulary of the controlled language. Several controlled languages with tool support have been proposed for RE. The most recent ones are Attempto Controlled English (ACE) by Fuchs and Schwitter [21,22] and the CREWS Scenario Authoring Guidelines by Ben Achour [5]

5.2 Documentation: Providing More Context Information

Comments, rationales, fit criteria, test cases, inverse requirements, and traceability information support the strategy of providing more context information.

1. A *comment* can be used to explain the background of a requirement. A comment that would allow the reader to disambiguate Example 29 is:

(33) Comment: A non-friendly aircraft is acceptable as long as it has a known mission or is more than 5 minutes away.

2. A *rationale* describes why a requirement is needed. Like a comment, a rationale can help disambiguate a requirement.
3. A *fit criterion* describes a condition that a software product must fulfill in order to satisfy a requirement. Each fit criterion thus provides contextual information and leaves less room for interpretation.
4. A *test case*, a more elaborated form of a fit criterion, describes a possible input and its expected output explicitly. Possible test cases for Example 29 are:

   (34) *Input*: Friendly aircraft.

   Output: No alarm.

   Input: Non-friendly aircraft and enters airspace within 5 minutes.

   Output: Raise alarm.

5. An *inverse requirement* describes functionality that the software product does not perform. Inverse requirements are often misused to express non-functional requirements, e.g., the system must not lose user data, which is actually a reliability requirement. However, in its essence, an inverse requirement rules out possible interpretations of one or more functional requirements. Thus, the inverse requirement disambiguates the functional requirements. Consequently, a true inverse requirement has no test case. If one can derive a test case for an inverse requirement, the inverse requirement is probably actually a non-functional requirement. An inverse requirement specifically helps to reduce pragmatic ambiguity, generality, and vagueness, as in the following example:

   (35) Requirement: The vending machine offers refreshing drinks.

   Inverse requirement: The vending machine does not offer tea, coffee, and alcoholic drinks.

   Therefore, most likely, the vending machine offers soft drinks, i.e., carbonated drinks, fruit juices, and water. The inverse requirement reduces the ambiguity of refreshing drinks by eliminating some potential meanings.
6. *Traceability information* on the dependencies between requirements, i.e., requirements–requirements traceability, also helps to disambiguate a requirement, if the links help identify closely related requirements that provide enough contextual information [29].

Independent of these specific techniques for providing more context information, there is the simple idea of asking for the participation of all stakeholders in the RE process. With such participation, all stakeholders share the same contextual knowledge about requirements, and misinterpretations are thus reduced.

A tool called the Requirements Apprentice, developed by Reubenstein and Waters, tries to automatically disambiguate a new requirement by putting the

requirement into a relation with existing requirements using the tool's pattern matching algorithm [60]. The proposed disambiguation is echoed by the tool to the user for user feedback and selection.

More information on augmenting a requirement with a comment, a rationale, a fit criterion, and a test case, and on inverse requirements and traceability is offered by Sommerville and Sawyer [66] and by Robertson and Robertson [61].

5.3 Documentation: Setting Up Conventions for Interpretation

There are no specific techniques for the strategy of setting up conventions, because this strategy is very pragmatic. Following this strategy, a convention in the case of the example requirement could be:

> (36) When a logical condition has more than one of "and" or "or", then the precedence rules of predicate logic are to be followed.

This convention must be clear to both the writer and the reader. Otherwise, misinterpretations can occur. In Example 29 the requirement would have to be rewritten to communicate the correct interpretation.

> (37) An aircraft that is non-friendly and has an unknown mission or **that is non-friendly and has** the potential to enter restricted airspace within 5 minutes shall raise an alert.

The bold-faced text was added to force the correct meaning on the assumption that and binds tighter than or.

5.4 Validation: Tools for Detecting Potential Ambiguities

While an automated tool cannot be relied on to find all and only ambiguities in a text, a tool can be built to assist a human reader in her task of finding ambiguities, perhaps by doing the more clerical parts of the task [38,26,27,17,54]. There are at least three flavors of tools, parser-based, pattern-matching, and semantic-network-based.

A parser-based tool such as CIRCE [26] attempts to parse the subject sentences to identify the component parts. Certainly the existence of more than one parse is a signal of an ambiguity. However, even if one is not interested in the existence of multiple parses, one might be interested in pattern matching. Identifying an instance of many of the patterns in pattern-based approaches assumes that the reader knows the parts of speech of words, and knowing the part of speech of a word instance requires knowing the parse of the containing sentence.

A pattern-matching tool searches for instances of any of a set of particular words, phrases, and even lexical affinities. For example, NASA has developed a

pattern matching tool for checking SRSs for particular patterns of imprecision [70]. Using such a tool to find instances of ambiguity patterns means building a dictionary of words, phrases, and lexical affinities [52] that are used in the patterns. While a phrase is a sequence of words, a lexical affinity is a set of words, generally within one sentence, that are separated by no more than some threshold of words apart from each other. For example, it is clear that there cannot be an ambiguity regarding the use of only unless the word only appears. Such a tool could not say for sure that a given use of only is ambiguous. However, just the tool's pointing out all occurrences of only to a human user, who then manually checks each one, should be a boon over relying on the user to find all occurrences of the only ambiguity. Similarly, one cannot have an ambiguity regarding the precedence of and and or unless an and and an or appear within a lexical affinity.

LOLITA [54,23] is a semantic-network-based tool, in that it is a collection of analysis applications built around a semantic network of linguistic and world knowledge. These applications can be arranged in a pipeline so that the output of any analysis can be used as input to one or more other analyses. LOLITA is in fact built out of two main internal applications:

1. an analyzer for translating from text to an internal logical form and
2. a generator for translating from the internal logical form to text.

More complex applications, such as natural language summarization, natural language translation, natural language querying, and a Chinese language tutor have been built on top of LOLITA. Since LOLITA contains a text analyzer, LOLITA can be used to build apparently parser-based applications. Since the results LOLITA's text analysis can be subjected to additional analysis, LOLITA can be used to build apparently pattern-matching applications.

Mich and Garigliano, in explaining an ambiguity finder based on their tool LOLITA [54,55], classify ambiguities into two types

1. semantic ambiguity, concerning the meanings of a word or phrase and
2. syntactic ambiguity, concerning the grammatical roles played by words in sentences and the grammatical structures of these sentences.

The word bank can be ambiguous in both senses. As a noun, bank can mean

- an edge of a river and
- a place in which one keeps her money,

possibly among others. Moreover, the word bank can be both a noun and a verb, meaning

- a place in which one keeps her money, and
- the act of dealing with a bank, as a noun, about money.

Mich and Garigliano propose algorithms to calculate for a text indices of ambiguity, both syntactic and semantic, based on the words that appear in the text.

- *semantic ambiguity*: a function of the number of possible meanings of the word
- *weighted semantic ambiguity*: a function of the number of possible meanings of the word weighted by the frequency of the word
- *syntactic ambiguity*: function of the number of possible syntactic roles of the word
- *weighted semantic ambiguity*: a function of the number of possible syntactic roles of the word weighted by the frequency of the word

They calculate these indices using the semantic network, i.e., the knowledge base, of their natural language processing system LOLITA and the outputs produced by LOLITA's text parser. The larger the dictionary used by the tool, the more meanings can be found for any given word. Weighted ambiguity measures can be used to help choose from among these many meanings.

The indices generated from a text may be used to rapidly signal to the user terms, phrases, and sentences that potentially have several interpretations. Armed with these signals, a requirements analyst may investigate the potentially offending sentences for actual multiple interpretations.

Examples of situations in which the measures can help include:

- When a questionnaire or interview is being designed, the questions can be tested to reduce the amount of ambiguity in them and to increase the chances that all being questioned will understand all questions in the same way, and thus that it is possible to meaningfully compare different persons' answers to the same question.
- When a Web page is being designed, the text of the page can be tested to reduce the amount of ambiguity in it and to increase the chances that all viewers will understand the same from the page.

It does not often happen that it is necessary to evaluate the ambiguity of isolated words. However, a practical application of the measure of lexical ambiguity of single words is to assess the menu commands of Netscape. To evaluate the semantic and syntactic ambiguity of the names of the commands, Mich used two different systems, LOLITA and Wordnet, a Lexical Database for English from the Cognitive Science Laboratory, at Princeton University [18], available at www.cogsci.princeton.edu/~wn/w3wn.html.

The lessons she learned from the experiment include:

1. Command names with the highest values of semantic ambiguity need to be examined for possibly changing them to less ambiguous names.

2. However, the values can be surprising and misleading, and should be used with caution.
3. Moreover, they remind us of the role of well-chosen graphic icons in disambiguating these terms.
4. Having command names of mixed grammatical type, i.e., some being nouns and some being verbs, is confusing and should be avoided in favor of making them all verbs.
5. The larger the dictionary used by the tool, the more meanings can be found for any given word.

5.5 Summary

Generally, ambiguity and lack of precision are major problems of natural language requirements. Thus, these problems should be addressed during all requirements engineering activities, from elicitation through validation. During documentation, a combination of all three strategies should be followed to help minimize ambiguity.

6. CONCLUSIONS AND PRACTICAL IMPLICATIONS

This chapter has surveyed the phenomenon of ambiguity in natural language requirements specifications. Since all specifications, even formal specifications, start from ideas expressed in natural language, the inherent ambiguity of natural language is inescapable in requirements engineering. After reminding the reader of some key issues relevant to requirements engineering, including the costs to fix a requirements error, the chapter gives a variety of definitions of ambiguity, namely dictionary, linguistic, and software engineering definitions. A conceptual model of software engineering ambiguities that offers a hope of systematic treatment of ambiguities in requirements documents is offered. Finally, a number of approaches to avoid writing ambiguities or to detect them in the writing are described.

The definitions and examples of ambiguity provided in Section 3 can be used by an RE practitioner to raise his awareness of the various facets of ambiguity. In our experience, everyone is aware of the potential for ambiguity, but few appreciate how much there can be, and are simply not aware of how ambiguous writing in software engineering, including requirements specifications, can be. Once ambiguity is recognized as a problem in a particular domain, then the techniques of Section 5 can be used to avoid and detect ambiguous specification in the domain.

In particular, the techniques described in Section 5 can be used by a practicing requirements writer or inspector to make his work easier. The requirements writer can use this material to help avoid introduction of ambiguities into specifications he is writing. The requirements inspector can use this material to help detect

ambiguities in specifications he is inspecting. The writer should build a check list of his recurring ambiguities, found repeatedly by inspectors, in order to be able to reduce their incidence in future specifications he writes. The inspector should build a check list of recurring ambiguities found in the specifications of each writer whose work he inspects, in order to improve his ambiguity detection effectiveness.

The approach discussed in Section 4 amounts to an advanced approach to explore the specific types of ambiguities that appear in the documents produced during software engineering, particularly when the impact of ambiguities can be large, e.g. in subcontracting.

Even with all the systematic techniques and modeling, avoiding and detecting ambiguities is at best an art. Fundamentally and ultimately, an ambiguity is anything that causes different people to understand differently; unfortunately, the set of people that happen to examine a document may just not have a person that detects an understanding that demonstrates an ambiguity. Therefore, it will be necessary to improve our skills at avoiding and detecting ambiguities. We need to teach requirements engineers to write unambiguously. We need to teach requirements engineers to spot hidden ambiguities.

Additionally, it will be necessary to validate all of the described techniques and tools for effectiveness in industrial settings and to do the same for all techniques and tools yet to be invented.

Finally, note that this chapter was written very carefully, with every effort paid to following our own advice and avoiding ambiguities. Indeed, the reader may have noted some unusual word order, especially with any word whose position in its containing sentence affects its meaning, e.g., also and only. A careful look at instances of these words will show that we are indeed following our own advice. However, despite our best efforts, it is possible that ambiguities remain. This point was driven home to us when we noted an ambiguity in a sentence as we were producing the final version. We fixed that problem, but just as with bugs, there is no last ambiguity!

ACKNOWLEDGMENTS

D.M. Berry's work was supported in parts by RENOIR, Requirements Engineering Network Of International cooperating Research groups, a European Union/ESPRIT network of excellence, project number: 20.800 (1996-2000), a University of Waterloo Startup Grant, and by NSERC grant NSERC-RGPIN227055-00.

REFERENCES

[1] *Merriam-Webster's Collegiate Dictionary,* Tenth Edition. Springfield, MA: Merriam-Webster, 1998. http://www.m-w.com/dictionary.htm

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

[3] Bach, K. "Ambiguity". In *Routledge Encyclopedia of Philosophy,* E. Craig and L. Floridi, eds. London, UK: Routledge, 1998.

[4] Basili, V., Caldiera, G., Lanubile, F., and Shull, F. Studies in Reading Techniques. Proceedings of the 21st Annual Software Engineering Workshop; 1996 December; Goddard Space Flight Center, Greenbelt, Maryland, USA. Software Engineering Laboratory Series, SEL-96-002, 1996.

[5] Ben Achour, C. Guiding Scenario Authoring. Proceedings of the 8th European-Japanese Conference on Information Modeling and Knowledge Bases; 1998 May 25–29; Vamala, Finland. IOS Press, 1999.

[6] Berry, D.M. The Importance of Ignorance in Requirements Engineering. Journal of Systems and Software 1995; 28(2): 179–184.

[7] Boehm, B.W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[8] Bostrom, R.P. Succesful Application of Communication Techniques to Improve the Systems Development Process. Information & Management 1989; 16: 279–295.

[9] Buley, E.R., Moore, L.J., and Owess, M.F. B5 (SRS/IRS) Specification Guidelines. Technical Report M88-57, ESD-TR-88-337, MITRE, Bedford, MA, USA, December 1988.

[10] Cruse, D.A. *Lexical Semantics*. Cambridge, UK: Cambridge Textbooks in Linguistics, Cambridge University Press, 1986.

[11] Dart, S.A. Spectrum of Functionality in Configuration Management Systems. Technical Report CMU/SEI-90-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, December 1990.

[12] Davis, A. *Software Requirements: Objects, Functions, and States*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[13] Denger, C. High Quality Requirements Specifications for Embedded Systems through Authoring Rules and Language Patterns. M.Sc. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 2002.

[14] Easterbrook, S. and Nuseibeh, B. Managing Inconsistencies in an Evolving Specification. Proceedings of the Second IEEE International Symposium on Requirements Engineering; 1995 March 27-29; York, UK. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[15] Easterbrook, S. and Nuseibeh, B. Using ViewPoints for Inconsistency Management. Software Engineering Journal 1996; 11(1): 31–43.

[16] Easterbrook, S.M. and Callahan, J.R. Formal Methods for Verification and Validation of Partial Specifications: A Case Study. Journal Systems and Software 1998; 40(3): 199–210.

[17] Fabbrini, F., Fusani, M., Gnesi, G., and Lami, G. An Automatic Quality Evaluation for Natural Language Requirements. Proceedings of REFSQ'2001, Seventh International Workshop on RE: Foundation for Software Quality; 2001 June 4–5; Interlaken, Switzerland.

[18] Fellbaum, C. *WordNet, An Electronic Lexical Database*. Cambridge, MA: MIT Press, 1998.

[19] Fowler, H.W. *A Dictionary of Modern English Usage*, Second Edition. Oxford University, Oxford, UK, 1965.

[20] Freedman, D.P. and Weinberg, G.M. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. New York, NY: Dorset House, 1990.

[21] Fuchs, N.E. and Schwitter, R.:Attempto Controlled English (ACE). Proceedings of CLAW'96, First International Workshop on Controlled Language Applications; 1996 March; Belgium.

[22] Fuchs, N.E., Schwertel, U., and Schwitter, R. Attempto Controlled English (ACE) Language Manual Version 3.0. Technical Report Nr. 99.03, Institut für Informatik der Universität Zürich, Zürich, Switzerland, 1999.

[23] Garigliano, R., Urbanowicz, A., and Nettleton, D.J. Description of the LOLITA System as Used in MUC-7. Proceedings of MUC-7, Message Understanding Conference; 1998; University of Durham.

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

[25] Gause, D.C. User DRIVEN Design—The Luxury that has Become a Necessity, A Workshop in Full Life-Cycle Requirements Management. Tutorial T7 in ICRE 2000, Fourth IEEE International Conference on Requirements Engineering; 2000 June 19-23; Schaumberg, IL.

[26] Gervasi, V. Environment Support for Requirements Writing and Analysis. Ph.D. Dissertation, TD-3/00, Dipartamento di Informatica, Università di Pisa, Pisa, Italy, 2000.

[27] Gervasi, V. and Nuseibeh, B. Lightweight Validation of Natural Language Requirements. Proceedings of ICRE'2000, Fourth IEEE International Conference on Requirements Engineering; 2000 June 19–23; Schaumburg, IL. Los Alamitos, CA: IEEE Computer Society Press, 2000.

[28] Goguen, J.A. "Requirements Engineering as the Reconciliation of Technical and Social Issues." In *Requirements Engineering: Social and Technical Issues,* J.A. Goguen and M. Jirotka, eds. London, UK: Academic Press, 1994.

[29] Gotel, O.C.Z. and Finkelstein, A.C.W. An Analysis of the Requirements Traceability Problem. Proceedings of the IEEE International Conference on Requirements Engineering; 1994; Colorado Springs, CO. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[30] Gray, J. Collection of Ambiguous or Inconsistent/Incomplete Statements. Nashville, TN: Vanderbilt University, 2000. http://www.vuse.vanderbilt.edu/~jgray/ambig.html

[31] Gunter, C.A., Gunter, E.L., Jackson, M., and Zave, P. A Reference Model for Requirements and Specifications. IEEE Software 2000; 17(3): 37–43,

[32] Hadad, G., Kaplan, G., Oliveros, A., and Leite, J.C.S.P. Integración de Escenarios con el Léxico Extendido del Lenguaje en la Elicitación de Requerimientos: Aplicación a un Caso Real. Revista de Informática Teórica e Aplicada (RITA) 2000, Brazil; 6(1): 77-104

[33] Harwell, R., Aslaksen, E., Hooks, I., Mengot, R., and Ptack, K. What is a Requirement? Proceedings of NCOSE, Third Annual International Symposium, National Council of Systems Engineers; 1993.

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

[35] Hirst, G. *Semantic Interpretation and the Resolution of Ambiguity. Studies in Natural Language Processing*. Cambridge, UK: Cambridge University Press, 1987.

[36] Hutchins, W.J. and Somers, H.L. *An Introduction to Machine Translation*. London: Academic Press, 1992.

[37] IEEE. *IEEE Recommended Practice for Software Requirements Specifications.* ANSI/IEEE Standard 830-1993. New York, NY: Institute of Electrical and Electronics Engineering, 1993.

[38] Ishihara, Y., Seki, H., and Kasami, T. A Translation Method from Natural Language Specifications into Formal Specifications Using Contextual Dependencies. Proceedings of the IEEE International Symposium on Requirements Engineering; 1993 January 4-6; San Diego, CA. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[39] Jackson, M. and Zave, P. Domain Descriptions. Proceedings of the International Symposium on Requirements Engineering; 1993 January 4-6; San Diego, CA. Los Alamitos, CA: IEEE Computer Society Press, 1993.

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

[41] Jackson, M.A. The Role of Architecture in Requirements Engineering. Proceedings of the IEEE International Conference on Requirements Engineering; 1994 April 18–22; Colorado Springs, CO. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[42] Jackson, M.A. Problems and Requirements. Proceedings of the Second IEEE International Symposium on Requirements Engineering; 1995 March; York, UK. Los Alamitos, CA: IEEE Computer Society Press, 1995.

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

[44] Kamsties, E., Berry, D.M., and Paech, B. Detecting Ambiguities in Requirements Documents Using Inspections. Proceedings of WISE'01, First Workshop on Inspection in Software Engineering; 2001 July 23; Paris, France.

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

[46] Kamsties, E. Surfacing Ambiguity in Natural Language Requirements, Ph.D. Dissertation, Fachbereich Informatik, Universität Kaiserslautern, Germany, also Volume 5 of *Ph.D. Theses in Experimental Software Engineering.* Fraunhofer IRB, Stuttgart, Germany: Verlag, 2001.

[47] Kaplan, G., Hadad, G., Oliveros, A., and Leite, J.C.S.P. Construcción de Escenarios a partir del Léxico Extendido del Lenguaje. Proceedings of SoST'97, Simposio en Tecnológia de Software, 26 Jornadas Argentinas de Informática y Investigación Operativa - SADIO.; 1997 August; Buenos Aires, Argentina.

[48] Kovitz, B.L. *Practical Software Requirements: A Manual of Content and Style.* Greenwich, CT: Manning, 1998.

[49] Leite, J.C.S.P. and Franco, A.P.M. A Strategy for Conceptual Model Acquisition. Proceedings of the IEEE International Symposium on Requirements Engineering; 1993 January; San Diego, CA. Los Alamitos, CA: IEEE Computer Society Press, 1993.

[50] Levinson, S. *Pragmatics.* Cambridge, UK: Cambridge University Press, 1983.

[51] Lyons, J. *Semantics I and II.* Cambridge, UK: Cambridge University Press, 1977.

[52] Maarek, Y.S. and Berry, D.M. The Use of Lexical Affinities in Requirements Extraction. Proceedings of the Fifth International Workshop on Software Specification and Design; 1989 May 19-20; Pittsburgh, PA.

[53] Mich, L., Franch, M. and Novi Inverardi, P. Requirements Analysis using linguistic tools: Results of an On-line Survey. Technical Report 66, Department of Computer and Management Sciences, Unversità di Trento, Trento, Italy, 2003. http://eprints.biblio.unitn.it/view/department/informaticas.html

[54] Mich, L. and Garigliano, R. Ambiguity Measures in Requirements Engineering. Proceedings of ICS2000, International Conference on Software Theory and Practice, Sixteenth IFIP World Computer Conference; 2000 August 21–24; Beijing, China. House of Electronics Industry, 2000.

[55] Mich, L. On the use of Ambiguity Measures in Requirements Analysis. Proceedings of NLDB'01, Applications of Natural Language to Information Systems, Sixth International Workshop; 2001 June 28–29; Madrid, Spain. Bonn: Springer, Lecture Notes in Informatics, 2001.

[56] Mullery, G. The Perfect Requirements Myth. Requirements Engineering Journal 1996; 1(2): 132–134.

[57] Oriel, J. Guide for Specification Writing for U.S. Government Engineers. Naval Air Warfare Center Training Systems Division (NAWCTSD), Orlando, FL, 1999. http://www.ntsc.navy.mil/Resources/Library/Acqguide/spec.htm

[58] Parnas, D.L., Asmis, G.J.K., and Madey, J. Assessment of Safety-Critical Software in Nuclear Power Plants. Nuclear Safety 1991; 32(2): 189–198.

[59] Parnas, D.L. Personal communication via electronic mail about bug-free programs, November 2002.

[60] Reubenstein, H.B. and Waters, R.C. The Requirements Apprentice: Automated Assistance for Requirements Acquisition. IEEE Transactions on Software Engineering 1991; 17(3): 226–240.

[61] Robertson, S. and Robertson, J. *Mastering the Requirements Process*. Harlow, England: Addison-Wesley, 1999.

[62] Rolland, C. and Proix, C. A Natural Language Approach for Requirements Engineering. Proceedings of CAiSE 1992, Conference on Advanced Information Systems Engineering; 1992 May 12–15; Manchester, UK.

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

[64] Ryser, J., Berner, S., and Glinz, M. SCENT Scenario Authoring Guidelines. Technical Report, University of Zuerich, Zuerich, Switzerland, 1998.

[65] 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 1992; 1(2): 188–204.

[66] Sommerville, I. and Sawyer, P. *Requirements Engineering, A Good Practice Guide*. Chichester, UK: John Wiley & Sons, 1997.

[67] Sorensen, R. Sharp Boundaries for Blobs. Philosophical Studies 1998; 91(3): 275–295.

[68] Strunk, W. and White, E.B. *The Elements of Style*, Third Edition. New York, NY: Macmillan, 1979.

[69] Walton, D. *Fallacies Arising from Ambiguity*. Applied Logic Series, Dordrecht, NL: Kluwer Academic, 1996.

[70] Wilson, W.M., Rosenberg, L.H., and Hyatt, L.E. Automated Analysis of Requirements Specifications. Proceedings of International Conference on Software Engineering, 1997 May 17-23; Boston, MA.

[71] Wing, J.M. A Study of 12 Specifications of the Library Problem. IEEE Software 1988; 5(4): 66–76.