RC24729 (W0901-059) January 16, 2009
Computer Science

IBM Research Report

Automated Inspection of Industrial Use Case Models Inferred from Textual Descriptions

Palani Kumanan, Amit Paradkar, Avik Sinha, Stanley M. Sutton Jr.
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Automated Inspection of Industrial Use Case Models Inferred from Textual Descriptions

Palani Kumanan, Amit Paradkar, Avik Sinha, Stanley M. Sutton Jr.*

IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY, USA 10598
{palanik,paradkar,aviksinha,suttons}@us.ibm.com

Abstract. Use cases are increasingly important for requirements capture. They feed into downstream activities such as requirements specification and test generation. The quality of use cases is thus a concern, and there are numerous guidelines for use case style and content. However, most use cases are written in (possibly structured) natural language, and their characteristics and quality can vary widely.
We have developed an approach for automated inspection of use cases based on the construction and analysis of models of use cases. Use case texts are analyzed by a Linguistic Engine that produces an abstract model of the use case. The model records linguistic properties and is analyzed to evaluate additional properties, such as integrity, completeness, complexity, concurrency, dataflow, and others. Feedback is provided to the user in the form of error and warning messages. We also provide guidelines to facilitate the writing of linguistically analyzable use cases. Authors can iteratively refine their use cases to eliminate problems. Appropriately correct use cases can be used as input for flow-graph or test-case generation.
We describe a prototype IDE that implements this approach and report on experience in analyzing and refining sample groups of industrial use cases.

1 Introduction

Errors in requirement are a leading cause of poor quality software [1]. Further, requirements defects that make their way into the field could cost several times as much to correct as defects that were corrected close to the point of creation [2, 3]. These observations have led to a renewed emphasis on requirement definition as evinced by commercial success of vendors with products in requirements definition space such as Ravenflow (www.ravenflow.com) and iRise (www.iRise.com).

We are working on a holistic approach to requirements definition which attempts to synthesize different modalities of requirements elicitation such as use cases, business process models, sketches & storyboards, and glossaries. Our focus so far has been on analysis of textual descriptions of use cases. In [4], we described a pipeline architecture for analyzing use case text and demonstrated

* Contact Author

its effectiveness (measured in precision and recall) in identifying key concepts in use cases - such as Actors, action verbs, etc. In this paper, we report 1) guidelines to make use case descriptions amenable to textual analysis, 2) an automated inspection technique for models inferred from use case descriptions and & an environment which implements this technique, and 3) its effectiveness when applied to 22 industrial use cases.

Use cases are an increasingly popular way to specify software requirements [5]. A use case (or *system use case*) is generally considered to be a dialog-like description of interactions between a prospective system and one or more clients (called *Actors*)of the system, often including a human user. The descriptions are written in more or less natural language, possibly in a structured document. The focus of the description is on observable actions and effects. In the context of a software development process, use cases are typically developed at the early stages of requirements gathering and may serve as input into the later stages of requirements specification or test-case definition (among other possible uses). In our previous work we have developed a system for automatically generating test cases based on use cases written in a relatively formal style [6].

The usefulness of a use case depends on the way in which it is written, both in terms of style and content. In practice, a putative use case may be written in an arbitrary style and with arbitrary content. However, in order to be of practical value, a use case should generally be clear, consistent, correct, and complete (within its intended scope). Its style should be straightforward and simple. It should not contain extraneous material (that is, unrelated to the elucidation of system behavior) and it should avoid presupposing aspects of system implementation (that is, the use case should address what the system does, not how it does it).

To help assure that use cases are written in a useful way, various standards or guidelines have been proposed [5], and evaluated in academic setting [7, 8] (including one reported at ECOOP'2001 [9]). Examples of guidelines are that sentences should conform to certain simple grammatical patterns, that verbs should be active and have present tense, and that descriptions of the mental states of users should be avoided.

Quality assurance is as much an issue for use cases as it is for other artifacts in the software life cycle. Approaches that have been used to assess the quality of use cases include human review of use case texts [9, 8] and automated analysis of use case texts [7, 10, 11]. However, the technique reported in [7] works on a structured subset of the language which may preclude its applicability to industrial use cases which are typically written in *unconstrained* language, thus limiting its appeal to practitioners. Fantechi *et al.* [10] use a sentence analyzer to look for lexical and syntactic issues such as: ambiguities indicated by words like *naturally* or subjectivity indicated by words such as *similar* etc. The text analysis used in [11] seems to be the closest in spirit to ours. However, due to proprietary nature of the technology, any comparison can be based only on the publicly available whitepapers. Based on these whitepapers, we conjecture that the use case text analysis in [11] occurs at an individual sentence level and not

across the entire use case (or across a set of use cases). Furthermore, automated analysis of previous works may entail the construction of a model to serve as an internal representation of the text, but these models are generally not exposed to developers or available for processing separately from the analysis.

In this paper we demonstrate an automated inspection approach that goes beyond previous approaches by making a model of the use case a first-class representation. This model plays a role with respect to the use case text that is comparable to the role that an abstract syntax tree (AST) plays with respect to a program text. It exists and is accessible separately from the program text and is available as input for any subsequent processing steps of interest. These steps may include such things as generation of flow models or test cases, and they can be decoupled from the creation of the model. In this paper we focus primarily on the use of models of use cases as the basis for analysis in support of development activities. The general contribution of the work is to show that use cases, which are substantially informal documents, can be treated to a large extent like formal artifacts (such as code), with corresponding opportunities for automated processing and prospects for life cycle support in the form of integrated development environments (IDEs).

Lastly, most of the reported work on defects in use cases is from academic environments and very few, such as [10, 12], have reported experience with industrial use cases. Of the two, only Törner *et al.* report statistics on defects found through *manual* inspection of automotive use cases. Fantechi *et al.* only show examples of issues with the use case text. In this paper, we report on errors in industrial use cases found using an automated inspection tool.

The rest of this paper is organized as follows. In the next sections we discuss our motivation for the use of models of use cases and give an overview of the general process in which we envision that these models will be used. In Sections 4 and 5 we describe our text analysis and guidelines for writing analyzable use cases. Sections 6 and 7 then describe our use case meta-model and approach to model analysis. Section 8 gives an overview of a prototype IDE that we have built to implement the described approach. Section 9 gives an example of the approach, showing analysis results for a population of sample use cases, illustrating the application of the guidelines to these use cases, then reporting analysis results following application of the guidelines. Finally we discuss related work and present a summary, conclusions, and future work.

2 Motivation

Why build models of use cases and make these a focus for their use in software development? There are several reasons:

- They provide a structure for storing information about the use case, including not only the results of textual analysis (e.g., grammar and content) but also information from other sources (e.g., mark-up by users).
- They can provide a more abstract representation of the use case than is embodied in the text. For example, actions in a use case can be represented

without regard for the voice or tense in which they were originally written. (Not all grammatical information may be needed for all uses of a use case.)
- Models of use cases can be related to other models that may be available in the context of software development, such as domain models, glossaries, test plans, and so on. These relationships can be exploited for uses such as consistency checking and change propagation (for example, [13]).
- The models can be subject to analysis. Certain information, such as relationships between use cases, may not be readily evident in use case texts. Additionally, the availability of a persistent model separately from the texts can allow analysis of the use cases separately from the creation of the model and without requiring reanalysis of the texts.
- Models of use cases are better suited than text as input to many tools, such as for test-case generation [6].
- In ways that are analogous to the uses of ASTs, models of use cases can serve as the focal point of an IDE in which tools and services can be integrated to support use case development and other use case related tasks.

Analysis of models of use cases can address "non-linguistic" information such as structure, control flow, dataflow, and so on. As noted, it can relate use cases to external sources of information (such as domain models), and it can evaluate conditions that span multiple use cases (e.g., the mutual consistency of a set of use cases). There are a number of ways in which such analyses can be useful. These include the identification of concerns or problems in the use case (such as consistency or style errors), the collection of metrics (e.g., on use case size and complexity), the provision of feedback for use case refinement, the integration of use cases from different sources, and the verification of the suitability of use cases for use in downstream activities, both automated and manual.

3 Process Overview

In the approach we are proposing, models of use cases are built by automated analysis of use case texts, possibly supplemented with additional information. New use case texts are written from scratch or existing texts are obtained from appropriate sources. These texts are sometimes written or revised according to guidelines that promote their analyzability for purposes of constructing the models. These guidelines (discussed in detail in section 5) are comparable to "readability" guidelines for human consumers of use cases. Use case texts are iteratively refined until they can yield an acceptably correct and complete model. The models are then analyzed to report on issues of concern (such as stylistic properties for human consumers or integrity properties for flow-graph generators). The use cases may be further refined to address issues identified by model analysis (e.g., overly complex sentences, or dangling references to included use cases). Once a use case satisfies the relevant conditions, it can be made available to downstream activities (such as requirements specification or test-case generation). Details and examples of this process are discussed in the following sections.

4 Text Analysis

Primary to our approach to text analysis is a linguistic analysis engine (LE) that extracts information from natural language use case text to create an analysis-ready computer model (described further in section 6). The linguistic analysis engine consists of multiple configurable components (see [4] for details) integrated using the Unstructured Information Management Architecture (UIMA) [14]. UIMA is an open, industrial-strength, scalable and extensible platform for building analytic applications or search solutions that process text or other unstructured information. UIMA provides a simple but rich representation for unstructured information. It enables different analysis components to share and extend their results, independently of the nature of their analysis algorithms. Additionally, UIMA enables developers to compose and configure such components and to combine them with existing or third-party components.

The syntactic analysis for our LE is achieved by means of a shallow parser [15]. Unlike deep parsers, shallow parsers do not attempt to parse the entire text and, instead, they scan the text, on the surface, for known phrasal patterns. Such parsers yield very high accuracies in parsing text where the sentence structures do not vary widely. The text in natural language use case descriptions does not, *ideally*, vary widely with regards to structure [4] and therefore, the LE has a very high accuracy. We report in [4], that when measured on 57 in–field use cases, the average precision of our LE is 86.1% and the average recall is 91.3%. Since the shallow parsers do not attempt to parse the entire text, they are incredibly robust to noise (extra-grammatical information e.g., currency signs, programming language constructs, colloquial terms) in text [16]. The LE leverages noise tolerance of shallow parsers and deploys it effectively to filter noise from text. The noise filtering mechanism is more of a requirement than luxury for a LE designed to analyze use case text. Our analysis [4] show that on an average 62.5% of use cases had presence of noisy text. As reported in [4], LE's average effectiveness in filtering noise is 90.9%.

Additionally, the LE deploys a lexical processor (the component that tokenizes the text, assigns base form to the words and associates part of speech information to tokens) which embodies lexical knowledge not only of "*unconstrained*" English but also for over a hundred other languages (including German, French, Spanish, Italian, Russian and Chinese). This greatly facilitates adaptation of the LE to different languages and and its applicability to multiple application domains. Applicability to multiple domains, again, is more a necessity than a feature. Use cases can be used to describe applications from a variety of domains including and not limited to HEALTH CARE, AVIATION, FINANCE, NETWORKING and SERVICES.

The configurability and composability aspects of the UIMA pipeline lends our LE a unique advantage over other existing LEs that use traditional problem specific architectures [17, 7]. Since the current LE is developed with domain independence as primary design focus, it yields almost uniformly accurate text processing across various domains. However, certain domain specific analysis activities may require greater accuracy. The composability aspect is a great help

in retargeting the currently domain independent LE to a domain specific text. For instance, retargeting the lexical analysis to a health-care-domain use case can be easily accomplished by configuring the LE by swapping in a medical text-aware lexical processor.

5 Analyzability Guidelines and Directives

The choice of a particular implementation of the shallow parser and a domain independent lexical processor makes our LE yield a better analysis if the input text is written in accordance to some norms. These norms essentially ensure that the input text does not violate the key rules of "*unconstrained*" language that the POS tagger of the lexical processor and the shallow parser assume to hold.

The set of norms can be classified into two categories: the directives and the guidelines. The directives are those that the input text must always follow and the guidelines are the ones that may enhance the quality of analysis. In other words, if the text does not conform to the directives, the LE will fail in its analysis; however, if the text does not conform to the guidelines the LE will produce partial models and occasionally, even correct models.

5.1 Directives

1. *Ensure end of sentence markers are correct*: One key functionality of the lexical processor component of the LE is to identify sentence boundaries. However, since the components is domain agnostic, its algorithm to recognize sentence boundaries is based on punctuation marks such as periods, commas and semi-colons. Without proper sentence identification, the shallow parser and the other components would not function. Thus, it is mandatory to ensure that punctuation marks, especially end of sentence markers are not missing[1].
2. *Ensure absence of slashes*: Use of forward and backward slashes in the text confuses the tokenizer component yielding wrong analysis from LE. Thus it is mandatory to avoid slashes. Cleaning up of slashes can be accomplished by replacing slashes with representative words e.g. "*or*" or by putting the containing phrase with quotes e.g. "*C:/root folder*".
3. *Ensure no non-ASCII characters*: Either due to different character encodings by the differing text editors or copying and pasting activities, the input text gets corrupted with non–ASCII characters. Such characters are not understood by the current input stream reader leading to failure of the LE. Hence it is mandatory to clean up the text before inputting to the LE. This involves re-typing quotations, hyphens and parentheses in text; removing extra spaces; and removing un-identifiable characters from the text.
4. *Ensure absence of unwarranted capitalizations*: Since the lexical processor of the LE is domain agnostic, it has a separate component for disambiguating

[1] If there are extra punctuation marks, the LE is usually able to filter them out

Parts–of–Speech information for words. The component uses an algorithm called Robust Risk Minimization (RRM) [18], which has some disambiguation rules based on orthographic information of the words (e.g. Capital/ Small starting letter). It is therefore, important to ensure that there are no un-warranted capitalizations that may confuse POS disambiguation process.

5.2 Guidelines

1. *Avoid grammatical mistakes*: The shallow parser is quite robust in handling extra-grammatical phrases and grammatically in-correct information. However, grammatical mistakes can sometimes mislead the RRM based POS disambiguation process. Of special importance, is to ensure a correct form of the action verbs to reflect correct tense and number agreement with the subject.
2. *Enclos noun phrases in quotes*: It is rare but not extremely rare to find occasional failures of the RRM algorithm. Thus a word "*use*" in a phrase like "*use case*", may get tagged as a VERB and this may confuse the shallow parser. Thus for safety, it is advised to enclose such noun phrases within quotation marks.
3. *Use determiners when describing subjects and objects of action*: Once in a while the RRM confuses a 3rd person singular verb with a plural of a noun. For instance, the word "*checks*" in the sentence "*Customer checks account for balance.*" may get classified as a plural of the noun "*check*" and the word "*account*" will be classified as a verb. Consequently, the word "*account*" will be picked up as an action by the LE and not the word "*check*" (as may have been the original intention of the author). Since both interpretations of the sentence viz., *"customer"* <u>*checks*</u> *"account balance"* and the *"customer checks"* <u>*account*</u> *"balance"* are syntactically valid, there is a natural ambiguity. In the absence of any domain specific guidance, the RRM chooses between such interpretations based on some heuristics that may occasionally falter. This leaves the LE with a risk of mis-identifying the primary action. Adding a determiner in front of the object of the action and modifying the sentence to *customer checks the account balance* disambiguates such sentences and hence, use of determiners is advisable.
4. *Avoid hyphenated comments*: Some use case authors use hyphens in the text to qualify sentences. For instance, consider the sentence "*System returns cash– this will be a multiple of 20.*". Such a construct may confuse the LE's ability to identify actors from the text. Thus, for safety, hyphens may be replaced with modification clauses, e.g.,"*System returns cash, which will be a multiple of 20.*"
5. *Punctuate to denote end of clauses*: Occasionally boundaries of a clause may not be evident to a shallow parser and hence it is advisable to put commas to indicate them. For instance, in a sentence, "*The customer who has the card (credit or debit card) is allowed to enter the bank.*", the boundary of the modification clause viz.,"*who has the card*" may not be evident to the shallow parser (in this case due to existence of the parenthesized modification

of "*card*"). Such confusions may be avoided by introducing a comma and modifying the sentence to "*The customer who has the card (credit card), is allowed . . .*".

6. *Avoid parallelizing noun phrases*: To a human reader, a phrase like "*steps 1 and 2* " may imply "*step 1*" and "*step 2*". However, the shallow parser may occasionally mis-interpret such phrases; especially when there is a POS ambiguity in the leading word (as in the case of "*step*"). Therefore, such phrases should be re-written and parallelism is generally discouraged.
7. *Avoid usage of pronouns:* The LE has the ability to resolve pronouns by automatic identification of the co-referring nouns, but with 95.3% accuracy (see [4]). Therefore, occasionally (approximately in 1 in every 20 instances) a pronoun may get replaced with a wrong noun[2], leading to a mis-analyzed text. Hence, the usage of pronouns is not encouraged.

6 Model Overview

The text analysis described in Section 4 extracts a *model* from input use case texts. As the extracted model forms the basis for subsequent analysis and other development activities, the feasibility and results of those activities are closely affected by the expressivity and correctness of the model. From the point of view of analytics for text processing, correctness and expressiveness are inversely related. The more expressive the meta-model, the more difficult it is to develop analytics that consistently produce a correct and complete model. This is especially true when the domain of discourse varies, as it dose for use cases. On the other hand, it is counter-intuitive, and possibly counter-productive, to restrict the expressiveness of the model when applications may require an information-rich representation. Thus we are prepared to sacrifice some correctness for expressiveness. To compensate, the text analysis may ignore problematic parts of the text, the texts can be iteratively refined to eliminate errors, authors can manually update parts of the model, the models can accommodate inconsistent elements, and we enable users or applications to ignore irrelevant errors.

Figure 1 depicts our meta-model at a high level. It represents the information contained in typical use case descriptions (UCDs) along with contextual information. For deriving the meta-model, we collected 27 UCDs from practitioners in the industry and from examples in the published literature [7, 5]. The primary criteria for selection was that the UCD sentences were approximately in accordance to the recommended practices [5].

At the most abstract level, an application model contains a model of the context and a model of use cases. The context is described through actors and business items. Each use case in the use case model is related to a use case description (UCD). The sentences in a UCD express one or more actions initiated by some actor or *a*gent such as a system or its user. Each action may have a set of parameters that are *defined* – assigned a value or *used*. Our implementation

[2] To mitigate this issue, the IDE can offer the use case author a choice while resolving the pronouns

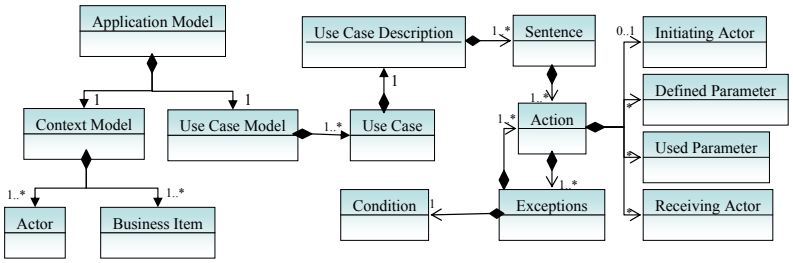

Fig. 1. Use Case Description Metamodel

of the model contains additional information that is not shown in the diagram, such as attributes of elements and additional relationships, roles, and subtypes.

Using the current metamodel we can capture information detailed enough for certain validation checks (described in following sections). We recognize that some applications may require additional information. The current meta-model is implemented using the Eclipse Modeling Framework (EMF) and is amenable to extension.

7 Model Analysis

Model analysis is performed to obtain information about a model of a use case and the underlying use case text that it represents. In turn that information may be used to guide, constrain, or contribute to subsequent development activities of various sorts. Abstractly, analyses take two different forms: reports and predicates.

Reports produce arbitrary output (typically text in some form). They may embody an arbitrary computation; these are presumed to be focused on a model of a use case but are not restricted to that. Reports are intended for information that may summarise a model or describe many collective elements in detail. Examples would be the collection of metrics or the gathering of data on the occurrence of errors (as used in this paper).

Predicates must produce a boolean result. Like reports, predicates may embody arbitrary calculations that are presumed to be focused on a model of a use case but are not restricted to that. Additionally, a predicate applies to a particular model element, which is its principal argument, and its result is conventionally associated with that element. Additionally, predicates can be assigned a severity level and interpreted as indicators of errors or other notable conditions. So, for example, a predicate may test whether as sentence has more than one action, violation of which which may yield a warning, or it may test whether a reference to a use case is defined, violation of which may yield an error.

Not every condition of interest for a use case can be evaluated automatically, but the set of interesting conditions that can be evaluated automatically is quite large (if not open-ended). We have implemented a range of predicates,

some of which exemplify commonly accepted standards for use case style and content, and some of which are of particular interest to us in relation to test-case generation. Some examples of these conditions are as follows:

- **Stylistic checks** for English sentences e.g., voice, use of actions of recognized kinds, use of anaphora.
- **Complexity checks** for the number of actors or actions in a statement, the number of updates to an item in a use case, and so on.
- **Completeness checks** of use case statements e.g., missing actors and actions, missing parameters.
- **Structural checks** for the model e.g., consistent use of aliases, dangling use case references.
- **Flow checks** for data and the control flow e.g., analysis for consistencies such as attempts to *use* items before they are *created*.
- **Ownership checks** that validate accessibility of data or appropriateness of actions relative to actors.
- **Concurrency-related checks**, e.g., for the occurrence of possibly concurrent actions or possibly non-serializable behaviors
- **Inter-model checks** to compare the actors and items referenced in a use case to element in an associated domain model

Some of the information evaluated by predicates is actually determined by text analysis when the model is constructed and represented as annotations on model elemenets (for example, the voice of actions). Other information is computed only when the predicate is evaluated, such as the complexity of sentences, data-flow properties, or aspects of model integrity.

Our analysis paradigm makes no assumptions about the conditions of interest, when those conditions should be evaluated, or what significance should be assigned to the results of evaluation. We believe these should be determined according to the opportunities and constraints arising from the environment, processes, products, and overall context of development.

To explore and validate the ideas discussed above, we have developed a prototype IDE that supports text analysis, model construction, model analysis, and other functions. Below we give an overview of that IDE and then present exaples that illustrate the application of the concepts using the IDE.

8 Prototype IDE

We have developed a prototype integrated development environment (IDE) as an enabler for the use case description process described in the Section 3. Figure 2 shows a screenshot of the IDE. The IDE facilitates development of relatively error-free use cases by assisting the author in elicitation, providing multiple views of the underlying model and providing feedback based on a set of validity checks defined on the model. The IDE also enables the invocation of additional tools on a use case model, including the generation of flow graphs and test cases based on use cases.

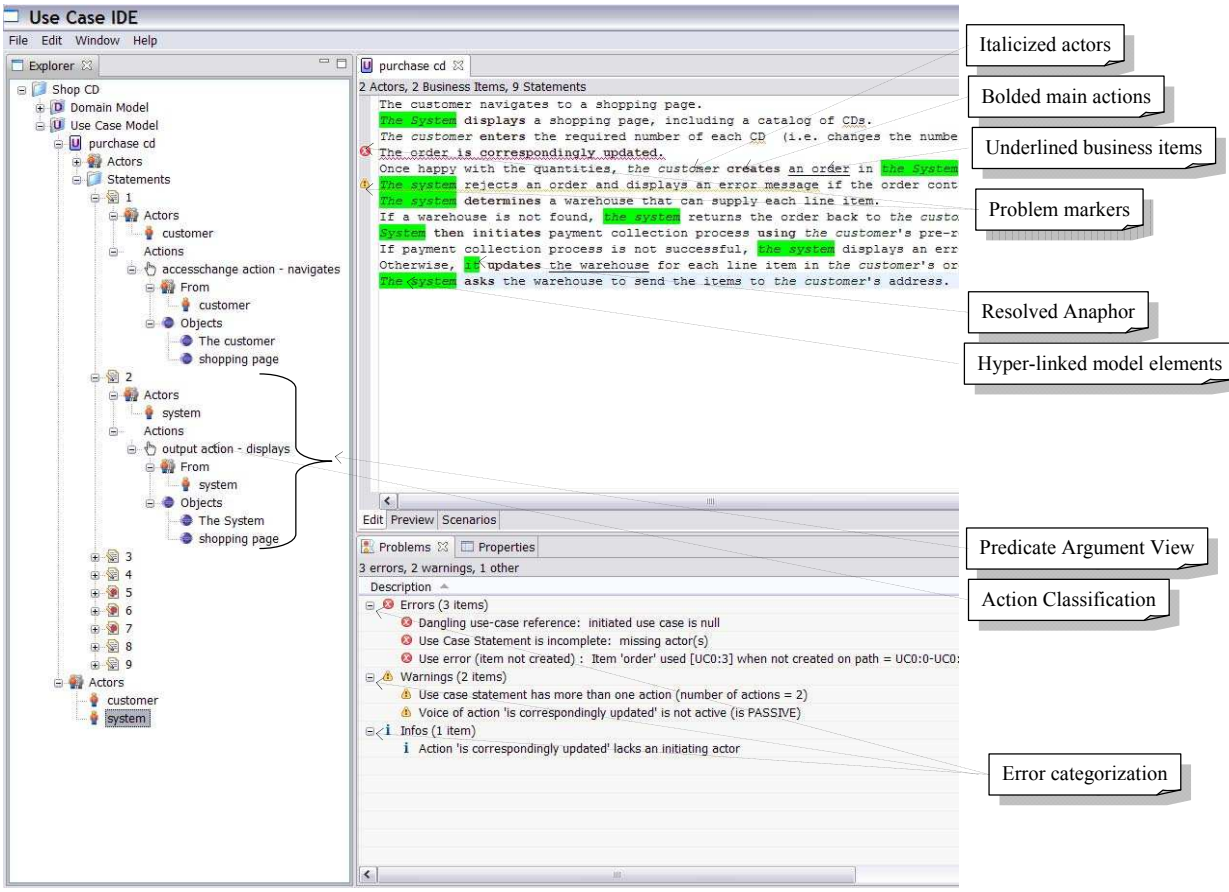

Fig. 2. The Online Analysis Environment

The IDE provides the author with the ability to create multiple use cases for a given application. The author starts the use case development process by creating a project and then adds use cases to the project. Each use case can be described using a text file. For elicitation of the textual use case description, the author is provided with a spell check backed text editor. On saving the textual description, the IDE invokes the linguistic engine and subsequently, the model analysis components. The model produced is analyzed for correctness and feedbacks are generated. Additionally, actors and business items identified in analyzed use cases are added to an associated domain model.

To assist the authors of a UCD in understanding the inferred conceptual model, the IDE provides multiple views of the model to the use case author. Notice the "*Explorer*" view on the left of the screenshot in Figure 2. The *Explorer* presents the model in a tree-view and helps the author to navigate or edit the

model. The use case is depicted as a series of sentence and each sentence as a collection of actions. Each action is displayed in the predicate argument structure where the arguments are its parameters and actors. The actions are classified into INPUT, OUTPUT, CREATE, READ, UPDATE, DELETE, DIRECT and INCLUDE.

The text editor itself acts as a view for the model by re-rendeing the input text to display model elements. Actors in the text are italicized and main actions are bolded. The model elements are hyper-linked. On selecting a model element on the *Explorer*, the corresponding text gets highlighted in the text editor. Notice that the pronoun "it" is hyper-linked to the actor "system" (thus showing a successful resolution of the pronoun).

Not shown in the screenshot are three additional views of the model that the IDE produces: the graphical view, the outlined view and the scenario view. In the graphical view, the IDE provides visualizations of the flow using the Business Process Modeling Notation (BPMN) [19]. In the outline view, each use case statement is listed in a enumerated list. The exceptions are listed as sub-bullets of the exception throwing statement and are marked by the exception condition. In the scenario view, the multiple scenarios of executing a use case are displayed under different tabs.

The "*Problems*" pane is shown in the screenshot; it lists the feedbacks based on the analysis of the model. A feedback includes a diagnostic of the problem and its severity. Notice the problems, their categories and their markers in the screenshot. The markers inform the users of the location in the input text that cause the failing tests. Clicking a problem, highlights fragments of text contributing to the error.

Not shown in the screenshot is a preferences page that allows for customization of predicate evaluation. Predicates are grouped into suites. Suites can be activated or deactivated individually, enabling or disabling their evaluation. Within a suite, the severity level of a each predicate can also be set individually.

The IDE is built as a rich-client application on Eclipse (http://www.eclipse.org). The Eclipse Modeling Framework (EMF, http://www.eclipse.org/emf/) is used to represent the use case model. Predicates and predicate suites are introduced into the IDE through Eclipse extension points that are defined as part of the IDE. The implementation of the text analysis component is described in Section 4.

9 Example

In this section we discuss an example of the use of the IDE to demonstrate concepts related to our approach to analysis of use-case models. The first subsection describes the population of use cases analyzed. The second reports on the initial results of model analysis, before the guidelines have been applied. The third subsection describes the application of the guidelines. The final subsection reports on the results of model analysis following application of the guidelines.

9.1 Example Use Cases

To illustrate the concepts advanced in this paper, we draw on two sets of example use cases obtained from industrial developers. The first set, "Set O", contains six use cases relating to an invoice management application. These were according to commonly recommended guidelines [5] such as using simple sentence structures, active voice, and generally describing the interactions between a user and a system. The use cases were written in plain text format and thus they ensured no special characters.

The second set of use cases, "Set N", contains sixteen use cases relating to a single E-commerce application. These use cases were written with relatively little guidance beyond a general overview of the use case concept. These use cases were originally written in a tabular format, e.g., with actors separated from actions. For text analysis we transcribed the text into a plain-text document. The transcription process may have introduced a set of non-ASCII characters.

9.2 Analysis of Given Use Cases

We analyzed the use cases in our example populations using a varied set of predicates. Table 1 lists representative predicates of particular interest. The predicates were chosen to reflect a variety of semantic concerns. Some of these are basic issues of correctness (e.g., statements with missing actions), some reflect commonly recommended stylistic constraints (e.g., use of active voice), and some reflect natural issues of model integrity (e.g., mutual consistency of actor aliases). The two dataflow predicates represent issues of computational correctness (these are examples from a larger group). A few are of particular interest to us as issues that affect the generation of test cases, such as the number of updates for an item in a use case and the possibility of concurrent behaviors. The setting of severity levels likewise reflects a particular set of priorities that we have–these can be adjusted as apporpriate to circumstances and objectives. These predicates and their severity levels are the primary basis for the statistics reported on use case analysis below.

Table 2 shows some summary data about the size of the example use case populations and the overall occurrences of errors and warnings as found by analysis of their models. The use cases in Set O average about eight statements each and have a moderate number of errors and warnings (about two and five per use case, respectively). Slightly more than half of the statements in these use cases have neither an error nor a warning. This may be expected since the authors were instructed to write in a use case appropriate style. The use cases in Set N average about eight statements each and have relatively many errors and warnings, on average more than one error per statement and about one warning per two statements. Nearly three quarters of the statements have at least one error or warning. This is consistent with a less directed style of use case writing.

Table 1. Representative Predicates for Use Case Analysis

Predicate Name	Applicability	Severity	Semantic Category
HasRecognizedActions	Statement	Error	Completeness
HasRecognizedActors	Statement	Error	Completeness
IsClassifiableAction	Action	Error	Correctness
IsUsedOnlyWhenDefined	Action	Error	Dataflow
AliasesMutuallyConsistent	Actor	Error	Integrity
NoDanglingUseCaseReference	Parameter	Error	Integrity
NoActorAliasing	Use Case	Warning	Style
HasOnlyActiveVoice	Statement	Warning	Style
NoDeleteWithoutUse	Action	Warning	Dataflow
NoConditionalStatements	Use Case	Warning	Style
NoObjectDefinedMoreThanOnce	Use Case	Warning	Complexity
HasActionsThatMayInterfere	Statement	Warning	Concurrency
DoesNotHaveMultipleActions	Statement	Warning	Complexity

Table 2. Summary of Size and Problem Occurrence in Given Use Cases

Group		# Use Cases	# Stmts	# Errs	# Warns	# Stmts w/Errs	# Stmts w/ Errs+Warns
Set O	Total	6	47	13	31	8 (17%)	22 (47%)
	Mean		7.8	2.2	5.2	1.3	3.7
Set N	Total	16	125	161	65	86 (66%)	95 (73%)
	Mean		7.8	10.1	4.1	5.4	5.9

Table 3 shows data on the kinds of errors that were found in the example use case populations. The relatively few errors in Set O are of various kinds. The occurrence of five dataflow errors reflects one statement in one use case where an item is used before being created, where that action affects five execution paths. The four dangling-use case references reflect an inconsistency in naming between the references and the intended use case file names. The "other warnings" reflect conditional statements, which we flag as possible use case exceptions.

The most striking thing about the kinds of errors for the use cases in Set N are the large numbers of unrecognized actors and actions. These go hand-in-hand, as often the identification of an actor depends on the identification of an action. The most common problem in identifying actions in these use cases is that the verbs in statements are often capitalized inappropriately (stemming from the original representation of these use cases in a tabular format). Some other factors that affect these problems are use of modal forms (not always a problem), sometimes complicated construction with passive forms, and other grammatical or punctuation issues. Compared to the Set O use cases, these use cases also contain a higher proportion of verbs that are not yet classifiable by our text analysis. As with the Set O use cases, a few occurrences of inappropriate data accesses create data flow problems on a larger number of execution paths. Likewise, most of the "other warnings" indicate conditional statements.

Table 3. Counts of Errors by Predicate Kind for Given Use Cases

Group		Missing Actor/ Action	Unclass. Action	Dataflow Errs/ Warns	Dangling Reference	Multiple Actions	Multiple Updates	Passive Voice	Other Errs/ Warns
Set O	Total	2/0	2	5/13	4	5	0	1	0/12
	Mean	0.3/0.0	0.3	0.6/1.6	0.5	0.6	0.0	0.1	0.0/2.0
Set N	Total	62/50	32	14/2	3	14	0	6	0/43
	Mean	3.9/3.1	2.0	0.9/0.1	0.2	0.9	0.0	0.4	0.0/2.7

9.3 Application of Guidelines

Following is a step-by-step description of the use of the guidelines presented in Section 5 to clean up text so as to improve analysis. The example is taken from one of the use cases in the example population for which we report analysis results; the application name has been modified for reasons of propriety.

In figure 3 we show a screenshot of a misanalyzed text. The as-is text is ridden with problems and needs to be cleaned according to the guidelines described in section 5. The analysis output (the misconstructed model) can be

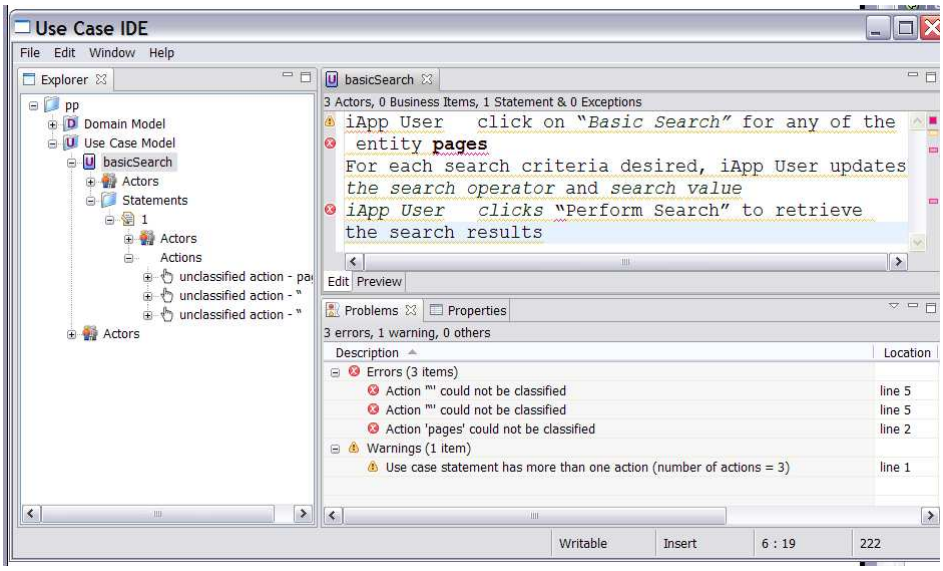

Fig. 3. Example Application of Guidelines–Screenshot 1

understood by looking at the "*Explorer*" pane of the screenshot. Notice that the misanalyzed text produces a model with just one use case statement containing three unclassified actions ("", " and *page*"). The primary cause for the failure is missing periods from the ends of the sentences. Additionally, the characters denoting the quotes are not ASCII-encoded.

Following Directives 1 and 3, we add a period at the end of sentences, remove extra spaces and re-type the quotation marks. On re-analysis, the LE produces the model as shown in Figure 4. The Explorer pane shows that not only did the the LE understand that there are three sentences, it also parsed accurately the second and the third sentences by identifying the correct actions, their voice, their initiators and their parameters. Notice, however, that the first sentence is still misanalyzed. This is because the word "*click* is not understood as an action. This is because the grammatical error (number disagreement of the verb "*click*" with the subject "*iApp User*") confuses the domain agnostic LE.

Following the Guideline 1 we correct the word "*click*" to its 3rd person singular form "*clicks*". This would correct the parse for the first sentence. Figure 5 displays the final outcome of the LE. By aiding in creating a text that is correctly analyzed, the guidelines also help in identifying a data flow problem. (As with program code, fixing some errors often exposes others.) Notice in the "*Problems*" pane of the figure the model analysis finds two data flow problems with the data elements "*search operator*" and "*search value*"– used before creation. Notice the source of the problems is the verb "*updates*", which is classified as an UPDATE action. The model analysis checks that an object is created before it is updated. In an application where the use case "*basicSearch*" is the only use case, this

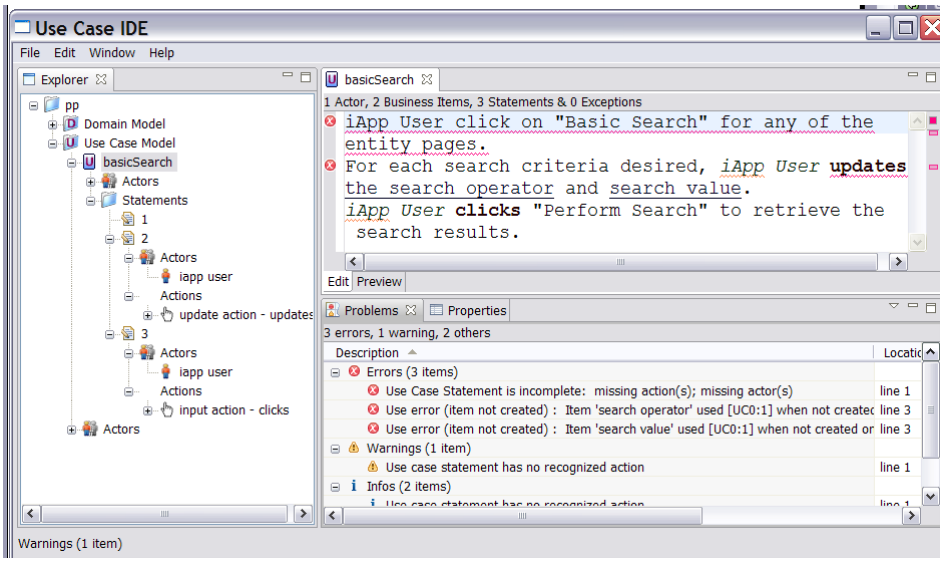

Fig. 4. Example Application of Guidelines–Screenshot 2

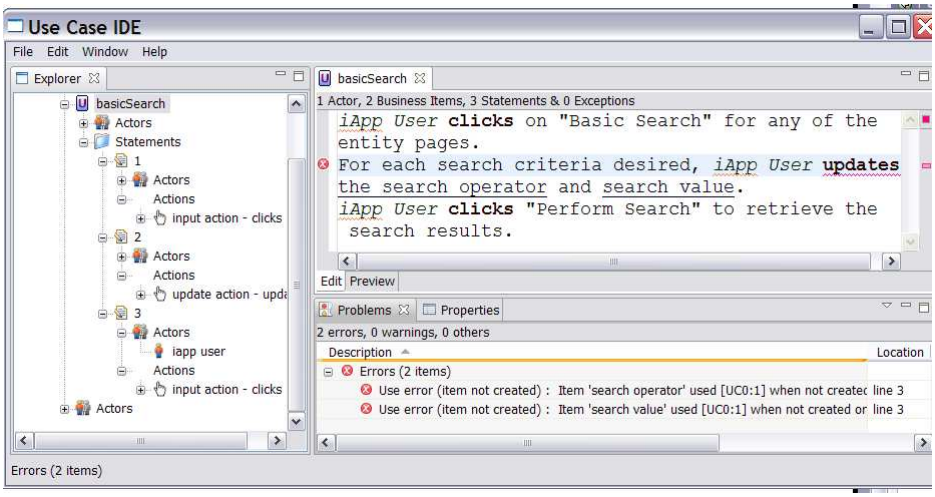

Fig. 5. Example Application of Guidelines–Screenshot 3

may be a source of problem. For designers, it might mean lack of specification of default value for "*search operator*" and "*search value*". The problem can be corrected by either choosing a different verb such as "*enters*" or adding an initialization statement before the *update* statement such as "*System initializes the search operator and the search value.*".

At this time we have no specific recommendations on the order in which the directives and guidelines should be applied–this is a subject for future research.

9.4 Analysis of Revised Use Cases

In this section we consider the change in numbers and kinds of errors that have resulted from the application of the guidelines for text analyzability. Table 4 parallels Table 2 but with values obtained after the application of the guidelines. For Set O, which was relatively error-free to begin with, the numbers have remained virtually unchanged. For Set N, which had relatively many errors, the numbers of errors and statements with errors have declined significantly, and the numbers for warnings have also declined slightly.

Table 4. Summary of Size and Problem Occurrence in Revised Use Cases

Group		# Use Cases	# Stmts	# Errs	# Warns	# Stmts w/Errs	# Stmts w/ Errs+Warns
Set O	Total	6	47	13	33	8 (17%)	22 (47%)
	Mean		7.8	2.2	5.5	1.3	3.7
Set N	Total	16	114	48	61	39 (34%)	55 (46%)
	Mean		7.1	3.0	3.8	2.4	3.5

Table 5 shows numbers of specific kinds of problems following revision. Comparing to Table 3, the numbers for Set O are very similar. In contrast, the numbers for Set N are greatly improved, with the total number of errors and warnings reduced by more than half. The numbers of missing actors and actions are drastically reduced, and the number of unclassified actions is also substantially lower. The numbers of other errors are more or less the same before and after revision. The better information regarding actions has lead to a shift in dataflow errors and allowed the recognition of some additional occurrence of passive voice.

While the application of the guidelines yielded a significant reduction in problems in the sample use case populations, it is important to appreciate that the process is not rigorous and does not necessarily lead monotonically to improvement in the use cases. The process of revising a use case text is manual, iterative and incremental, subjective, and naturally variable. Moreover, the results of revising a use case, like the initial writing of the use case, may be subject to further review and validation.

Table 5. Counts of Errors by Predicate Kind for Revised Use Cases

Group		Missing Actor/ Action	Unclass. Action	Dataflow Errs/ Warns	Dangling Reference	Multiple Actions	Multiple Updates	Passive Voice	Other Errs/ Warns
Set O	Total	2/0	2	8/12	4	6	0	1	0/12
	Mean	0.3/0.0	0.3	1.0/1.5	0.5	0.8	0.0	0.1	0.0/2.0
Set N	Total	10/3	19	13/4	3	14	0	10	0/33
	Mean	0.6/0.2	1.2	0.8/0.3	0.2	0.9	0.0	0.6	0.0/2.1

Our subjective analysis of the residual issues indicates that most of them (with the exception of few unclassified action verbs) are issues that violate one or more of the guidelines suggested for use case authoring [7, 9, 8] and would need intervention of the author to resolve. Of the kinds of problems that remain in our example, for instance,

- Sentences that still lack a recognized actor or action should be subject to further revision or (possibly) elimination. For example, the following sentence `This input amount is applied to all offer items to be created.` has missing actor
- Verbs that represent actions that cannot be categorized may need to be replaced by other verbs–or, if they should stand, then the action should be subject to particular review regarding issues of concern (e.g., dataflow, access, or concurrency issues). For example, in the sentence `"The shopper may refine the search to be performed"`, the verb `may refine` is not classified since it generally does not fit into any of the conversation categories for use case descriptions. However, in this particular example, it may indicate that the `shopper` is providing more details on some INPUT supplied previously. We are investigating the mechanism to incorporate the necessary analysis in our IDE.
- Statements in passive voice may be accepted or revised to active voice, depending on applicable style standards
- Similarly, statements that contain multiple actions may be split into multiple statements, depending on applicable style standards
- Dangling use case references can be repaired or eliminated
- Data flow errors can be resolved by the addition or removal of appropriate actions (e.g., adding an explicit create operation or removing an unnecessary delete operation)

Depending on the predicates of concern, the standards to which the use cases are initially written, and the priorities for the development activity, the process of revising use cases may vary significantly. In general, we do not expect that application of our guidelines will necessarily result in an error-free use case; rather, our goal is to obtain a use case is increasingly clean and usefully analyzable for non-textual problems and semantic concerns of the application and domain.

10 Related Work

In this section we compare our approach to related work along several dimensions of concern.

10.1 Structure of Use Cases

Many recommended or practiced approaches to the specification of use cases adopt the use of some sort of structured representation, often a template, table, or structured document [7, 5]. The different parts of the structured representation address different kinds of information, such as preconditions, normal flow, exception, and so on, although the usual representation is text. The text analysis in our Use Case IDE operates on unstructured text documents (effectively on text files). Information from structured formats can be accommodated by our text analysis if it is first transcribed into an unstructured document. (We have had to do this for some of the sets of use cases that we have analyzed.) Work is underway to enable our use case IDE to address structured representations of use cases.

10.2 Structure of Statements

It is commonly recommended that use-case statements have a straightforward structure. Some work in this area recommends the use of templates for use-case statements. For instance, the CREWS project [7] led to eight "content guidelines" that are templates sentence templates such as

- `<agent> <action> <agent>`
- `'If' <alternative assumption> 'then' <action>`
- `<agent> <'move' action><object> from <source> to <destination>`

10.3 Kinds of Conditions

There are many conditions of potential concern for use cases and these can be classified in several different ways: semantic domain, level of significance (or severity), and means of analysis (e.g., automated versus manual). Different workers have focused on different sets of conditions and categorized these in different ways. For example, [7] contains proposals for style guidelines and content guidelines. The content guidelines are actually templates for acceptable sentence structures. In our use case IDE, we have not required or recommended specific sentence structures, as these are immaterial for our purposes so long as a sentence is analyzable. However, the recommended structures should facilitate analysis by our use case IDE, which could in principle verify conformance to such sentence structures (although we have not addressed that). The style guidelines are mainly grammatical, linguistic, and structural, for example, use present tense and active voice, use consistent names, avoid anaphora, begin every statement

on a new line, number each statement consecutively, express alternate flows separately, and so on. Our use case IDE has the ability to automatically evaluate many of the linguistic concerns. It does not yet address structuring within a use case, although many of those concerns should also be amenable to automated verification. Certain issues, such as whether names are use consistently, cannot be verified automatically, although these may receive some automated assistance (such as presentation of names used for human review).

The authors in [12] have selected a set of conditions based on a review of prior literature (cited there). The broadly classify their conditions under completeness, correctness, consistency, readability, and unambiguity (with some additional concerns that might be put under a heading of appropriateness). We use a somewhat comparable set of categories for our conditions in Table 1: completeness, correctness, complexity, integrity, dataflow, concurrency, and style. Actually, neither they nor we define what we mean by our respective categories. Even so, there seems to be some overlap (e.g., completeness and consistency) with some significant differences (e.g., readability in their case, concurrency and dataflow in ours). This may reflect a difference in the expected mode of evaluation and intended application of the use cases. In [12] the conditions are evaluated manually and the use cases are intended for human consumption, whereas in our case the conditions are evaluated automatically and the intended use (or one of them) is the automated generation of test cases. Perhaps related to this, there is a significant difference in the specificity of the conditions in [12] and here. The scope of their conditions are suggested by representative questions, may be open ended, and may not be amenable to automatic verification. Example questions include "Does the main flow fail to achieve the goal/purpose?" and "Will the use case be understandable in 20 years?"[3] In contrast, the conditions we evaluate are generally more specific and all have a precise operational semantics. It should be noted that some of the conditions in [12] are simpler, such as whether a statement is missing an element. Also, automated analyses may contribute to some of the more complex evaluations, for example, complexity metrics may contribute to the assessment of whether a use case should be split up.

The conditions supported by [11] are of interest because those are also evaluated automatically. The approach used is to analyze a use-case text to build a flow model of the use case and then to reflect possible errors in terms of the flow model. The authors don't suggest a general classification of conditions, but the conditions they address are compatible with the classification we have adopted here. Specific conditions they consider include flow breaks, inconsistent names, missing actors for conditions, incomplete conditionals, unspecified means of communication, language errors (e.g., passive voice), and missing steps or process errors. Some of these are identified explicitly (e.g., passive voice), others depend on recognition by a human based on representation of the use-case flow (e.g., process errors). This is generally true of our work, as well, as our use case IDE enables some conditions to be evaluated directly while others must be assessed by the user based on inspection of the use case or inference from other analysis

[3] Strictly speaking, we don't know that use cases will be in use in 20 years.

results. The specific conditions addressed in [11] are generally similar to ones that we can evaluate in our IDE, although, as noted in Section 1 our analysis works across the sentences in both single and multiple use cases, whereas that in [11] seems to be limited to analyzing single sentences within a use case. Further, our IDE seems to address a greater number of conditions, to be more extensible in terms of the introduction of conditions, and more flexible in terms of the evaluation of conditions.

An interesting aspect of the style guidelines in some work (e.g., [7, 12]) is that they are often really composites of several simpler conditions, for example

> SG1 : write the UC normal course as a list of discrete actions in the form:
>
> `<action #> <action description>.`
>
> Each action description should start on a new line. Since each action is atomic, avoid sentences with more than two clauses [7].

We have tried to express our own guidelines (Section 5 in terms of focused rules (e.g., "Ensure no non-ASCII characters"). Also, we typically write predicates on relatively specific conditions, for example, testing for at least one action per statement or at most one action per statement. These can then be assessed individually or combined in various ways to serve various purposes. In our IDE, simple predicates can be referenced by composite predicates, the evaluation and severity level of predicates can be controlled individually, and predicates can be gathered into suites that represent different constellations of concerns.

11 Summary, Conclusions, and Future Work

We have developed an approach for the writing of quality use case descriptions. The approach has four main elements:

- Guidance in the writing of textual use-case descriptions
- Automated linguistic analysis of use-case texts
- Construction of abstract models of use cases based on linguistic analysis and other information
- Automated analysis of models of use cases for a customizable and extensible variety of quality properties

The approach is embodied in a prototype IDE (integrated development envorionment) built on Eclipse. This IDE supports the editing of use case texts, manages the invocation of analyses, provides feedback to authors based on analysis results, and generally facilitates the iterative refinement of use case texts. Analyses are represented as predicates and reports. Predicates are grouped into suites that can be activated and deactivated, and the severity level associated with each predicate can be set individually. In this way the IDE allows the quality criteria

to be customized and to evolve over time. The IDE also enables acceptable use cases to be provided as input to other activities, such as automated flow-graph construction or test-case generation.

The guidelines we offer are largely consistent with other guidelines in the literature but are also tailored to facilitate the writing of use cases that are amenable to automated analysis. We have applied our analysis to several dozen use cases obtained from several industrial sources. The style and quality of these use cases varies widely. As described in this paper, some of these use cases are more readily analyzed than others. As shown, application of the guidelines can greatly reduce the apparent errors in a use case and greatly facilitate its analysis. Repeated application with attentive human review can remove spurious errors and enable a focus on fundamental problems.

The literature contains many examples of guidelines that are not amenable to automated analysis, such as issues of readability, relevance, and certain aspects of correctness. While not able to resolve these in general, automated analysis may still be able to provide supporting information in some cases. Moreover, automated analysis of use cases, as for programs, can identify concerns that may be difficult or tedious for a human reader to catch, such as problems related to dangling references, dataflow, and concurrency. Thus we believe that the most general approach to assuring use-case quality will involve human review and automated analysis applied in concert.

Future work will involve enhancement of the linguistic analysis, addition to the set of predicates and reports, enrichment of the use-case model and analyses to better support downstream applications, and experimentation with use-case development processes. In particular, we are interested in experiments which evaluate the effectiveness of our use case inspection technique on software productivity and quality. Also, given our initial focus on creating test cases from use case descriptions, we would like to conduct experiments with test cases generated from the inspected use cases.

Acknowledgments

We thank Branimir Boguraev of IBM T. J. Watson Research Center for his help in the area of text analysis. We thank many colleagues at IBM and its client organizations for their contribution of example use cases.

References

1. Lutz, R.: Analyzing software requirements errors in safety-critical, embedded systems. In: RE'93. (1993)
2. Boehm, B.W., Papaccio, P.N.: Understanding and controlling software costs. IEEE Transactions on Software Engineering **14**(10) (1988) 1462–1477
3. Willis, R.R., et al.: Hughes aircrafts widespread deployment of a continuously improving software process. Technical Report Document CMU/SEI-98-TR-006, Software Engineering Institute, Pittaburgh, PA (1998)

4. Sinha, A., Paradkar, A., Kumanan, P., Boguraev, B.: A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In: DSN '09. (2009) submitted
5. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Boston, MA, USA (2000)
6. Kaplan, M., Klinger, T., Paradkar, A., Sinha, A., Williams, C., Yilmaz, C.: Less is more: A minimalistic approach to uml model-based conformance test generation. In: ICST '08. (2008) 82–91
7. Rolland, C., Achour, C.B.: Guiding the construction of textual use case specifications. Data Knowl. Eng. **25**(1-2) (1998) 125–160
8. Karl, C., Aurum, A., Jeffrey, R.: An experiment in inspecting the quality of use case descriptions. Journal of Research and Practice in Information Technology **36**(4) (2004)
9. Anda, B.C.D., Sjøberg, D.I.K., Jørgensen, M.: Quality and understandability of use case models. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference. (2001) 402–428
10. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Application of linguistic techniques for use case analysis. Requirements Engineering Journal **8**(3) (2003) 161–170
11. "RavenFlow Inc.": "www.ravenflow.com" (2008)
12. Törner, F., Ivarsson, M., Pettersson, F., Öhman, P.: Defects in automotive use cases. In: Int. Symposium on Empirical Soft. Eng. '06. (2006) 115–123
13. Sinha, A., Kaplan, M., Paradkar, A., Williams, C.: Requirements modeling and validation using bi-layer use case descriptions. In: MoDELS '08, Berlin, Heidelberg, Springer-Verlag (2008) 97–112
14. Ferrucci, D., Lally, A.: UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering **10**(4) (2004)
15. Boguraev, B.: Towards finite-state analysis of lexical cohesion. In: 3rd International Conference on Finite-State Methods for NLP, Liege, Belgium (2000)
16. Hammerton, J., Osborne, M., Armstrong, S., Daelemans, W.: Introduction to special issue on machine learning approaches to shallow parsing. J. Mach. Learn. Res. **2** (2002) 551–558
17. Fliedl, G., Kop, C., Mayr, H.C., Salbrechter, A., Vöhringer, J., Weber, G., Winkler, C.: Deriving static and dynamic concepts from software requirements using sophisticated tagging. Data Knowl. Eng. **61**(3) (2007) 433–448
18. Zhang, T., Damerau, F., Johnson, D.E.: Text chunking based on a generalization of Winnow. Jnl. of Machine Learning Research **2** (2002) 615–637
19. Group, O.M.: Business process modeling notation version 1.1. http://www.bpmn.org/Documents/BPMN1-1Specification.pdf