RC24877 (W0910-084) October 15, 2009
Computer Science

IBM Research Report

Text2Test: Automated Inspection of Natural Language Use Cases

Stanley M. Sutton Jr., Avik Sinha, Amit Paradkar
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

Text2Test: Automated Inspection of Natural Language Use Cases

Stanley M. Sutton Jr., Avik Sinha, Amit Paradkar
I.B.M. T. J. Watson Research Center
19 Skyline Drive,Hawthorne, NY, USA
{*suttons,aviksinha,paradkar*}*@us.ibm.com*

***Abstract*—The modularity and customer centric approach of use cases make them the preferred methods for requirement elicitation, especially in iterative software development processes as in agile programming. Numerous guidelines exist for use case style and content, but enforcing compliance to such guidelines in the industry currently requires specialized training and a strongly managed requirement elicitation process. However, often due to aggressive development schedules, organizations shy away from such extensive processes and end up capturing use cases in an ad-hoc fashion with little guidance. This results in poor quality use cases that are seldom fit for any downstream software activities.**

We have developed an approach for automated and "edit-time" inspection of use cases based on the construction and analysis of models of use cases. Our models contain linguistic properties of the use case text along with the functional properties of the system under discussion. In this paper, we present a suite of model analysis techniques that leverage such models to validate uses cases simultaneously for their style and content. Such model analysis techniques can be combined with a robust NLP techniques to develop integrated development environments for use case authoring, as we do in Text2Test. When used in an industrial setting, Text2Test resulted in better compliance of use cases, in enhanced productivity and, subsequently, in higher quality of test cases.

***Keywords*-Requirements, Use Cases, Testing, Automated Inspection, Analysis, Text2Test**

I. Introduction

Ever since they were introduced by Jacobson in [1], *use cases* have enjoyed popularity among business analysts (BA) and requirement analysts (RA). BAs and RAs like use cases because of their modularity, simplicity and user centric approach. In practice, due to the need of BA's and RA's to interact continuously with the customer, use case descriptions are written in natural language. In the past, this has been a hurdle for computer aided software engineering (CASE) that aim to leverage use case descriptions for automation of downstream activities such as iteration planning, requirement validation or test case generation. As a remedy, researchers have usually suggested use of formal notations with pre-defined semantics for writing use cases. Such notations, while readily consumable by the case activities for functional analysis, create hurdles for BAs and RAs in using use cases for communicating to customers. Natural language, therefore, still remains the preferred way of writing use cases. While there are guidelines that target communication and functional properties of NL use cases, automatic enforcement of such guidelines are largely missing. For high quality use cases we need automated analysis techniques that validate both the linguistic style and the functional content of the use cases.

Recent advances in Natural Language Processing (NLP) techniques have led to a few high accuracy technologies for interpreting natural language use cases [2], [3], [4]. In fact, our home-grown technology discussed in [4], when measured on 57 industrial use cases, demonstrated a very high average precision of 86.1% and an excellent average recall of 91.3%. Such a technology enables use case contents to be extracted and represented in a model. 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. An AST-like representation simultaneously preserves syntactic and functional properties of use cases. It exists and is accessible separately from the program text and is available as input for any subsequent processing steps. 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 this 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 IDEs. Specifically, in this paper, we report on 1) a AST-like-model based approach to use case validation; 2) Text2Test, an integrated development environment (IDE) for use case authoring; and 3) its effectiveness on automated test case generation when applied in an industrial setting.

II. Background, Related Work and Motivation

Use cases serve two primary needs in a software development process, especially in an agile software development environment. They serve as a written contract between developers and customers and they serve as requirement documents for down-stream processes. Superior quality of use cases should therefore serve both the communication value and the functional value.

In order to be of any communication value, language of a use case should generally be clear, consistent and correct. 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). In the past, researchers have attempted to address these concerns via standards or guidelines [5], [2], [6], [7], [8]. Common examples of such 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. With the exception of work by [2] these guidelines have never been automatically enforced during use case editing. We believe enforcing such guidelines during use case editing are important and are of value to authors and consumers of the use cases.

Functionally use cases should represent scenario based requirement specification in a consistent and unambiguous fashion. Approaches related to functional analysis of use cases include human review of use case texts [7], [6] and automated analysis of use case texts [2], [9], [10]. However, the technique reported in [2] 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.* [9] 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 analysis used in [10] 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 white-papers. Based on these white-papers, we conjecture that the use case analysis in [10] 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.

Lastly, most of the reported work on defects in use cases is from academic environments and very few, such as [9], [11], 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. One of our other motivation is assess the effectiveness of our technique in industrial use cases.

III. Methodology

Our approach to assuring the quality of use cases is centered on persistent, shared models. We perform a linguistic analysis of the use-case text to construct the model, then analyze the model consistency and other properties. We have adopted a model-centered approach for several reasons:

- The models provide a structure for storing information about the use case, including both results of textual analysis and information from other sources (e.g., mark-up by users).
- Models 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 is needed for all purposes.)
- 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, [12]).
- 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 [13].
- 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 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 downstream applications, both automated and manual.

The core of our approach to use case specification is an interactive write-analyze-revise cycle that is analogous to the edit-compile-debug process familiar from the development of code. Abstractly, the author of the use case enters natural language text into the system through an editor (or via an import facility). The system invokes the text analysis engine to produce a model of the use case (analogous to compiling a program unit to produce an AST). The system automatically analyzes the model of the use case, in conjunction with models from associated use cases, and reports on problems found. The problems relate to both style and content of the use cases. The author can then refine the use case text ("debug" it) so as to improve its quality or suitability for various downstream uses.

In contrast to the usual practice with code, however, we envision that use cases may be developed or modified by people in different roles and at different stages of the life cycle. For instance, a Business Analyst may write use cases that are more business-oriented and that have a less formal style, while a Test Designer may write use cases (possibly refining a Business Analyst's use cases) that are more system oriented and that have a more formal style. To support the different requirements that these different users may have for use case quality and consistency, we allow the predicates by which use cases are assessed to be varied according to context. So, for instance, a Software Designer's use cases may have to meet certain technical conditions for correctness of data flow, while the Business Anaylst may care nothing about data flow but may be concerned with understandability.

Additionally, we expect that use cases will be put to different uses by people in different roles at different points in the life cycle. For instance, the Business Analyst may want to generate process diagrams for review by domain specialists, whereas the Test Designer may have responsibility for generating test case from the use cases. The ability to support a variety of downstream uses is one of the main motivations for parsing the use case text into an abstract model. Along with our prototype we have developed tools to generate process models, generate test cases, and create problem summary reports.

IV. Text2Test

To give a more concrete view of our approach, let us describe Text2Test, a platform for authoring use cases. Text2Test enables an edit-time analysis and feedback system for use case authors. Figure 1 shows a screenshot of Text2Test. The platform provides a rich-text editor with spell checking to support use case entry. As mentioned before, the core of our approach to use case specification is an interactive write-analyze-revise cycle . Text2Test supports such a cycle by providing online analysis and feedback. The author starts by creating a project in Text2Test. To add use cases to the project, the author of the use case enters natural language text into the system through an editor (or via an import facility or via direct graphical editing). The system invokes the text analysis engine to produce the model of the use case (analogous to compiling a program unit to produce an AST). The analysis is triggered automatically on creation or update of this model. Issues found with the underlying model are listed in the "*Problems*" pane, as 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 the fragments of text contributing to the error. Problem markers are also used to generate corresponding error annotations in the editor. Using this feedback mechanism, the author can then iteratively refine the use case text and improve its quality or suitability for various downstream uses.

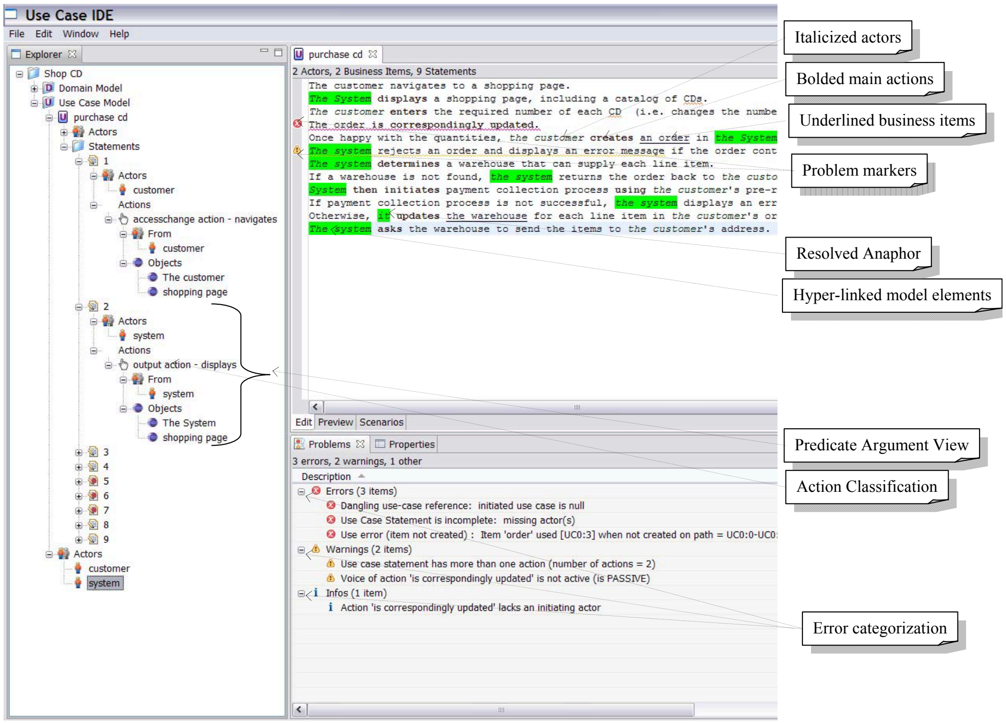
The underlying model is presented back to the user in multiple views, *e.g.*, the graphical view, the outlined view, the scenario view, and the properties view. In the graphical view, Text2Test provides visualizations of the flow using the Business Process Modeling Notation (BPMN) [14]. In the outline view, each use case action is listed in a enumerated list. In the scenario view, the multiple scenarios of executing a use case are displayed under different tabs. In the properties view, properties are listed for elements selected in the *Explorer*. The purpose of providing a multitude of views, is to facilitate multiple stakeholders (BAs, RAs, Customer, Designer, Developer, Tester, etc.) understand the abstract model.

A. Stake-holder specific feedback

Use cases need to be validated for different purposes throughout an application life-cycle. For instance, while early in the life-cycle a Business Analyst may be interested in checking if the use case specification is complete, he may not be interested in knowing the data flow errors in the specification. Such errors, however, could be of interest to a Test Designer later in the life-cycle. Customized suites of predicates can be chosen selectively by the author.

Figure 2 shows the feedbacks resulting from different configurations of predicates applied to the same use case. Notice in Figure 2(a), that when configured for a hypothetical Business Analyst, the prototype IDE returns three errors related to problems with parsing or classifying sentence elements, and it gives three warnings related to stylistic concerns such as use of conditionals, complex sentences (multiple actions), and passive voice. When configured for a hypothetical Test Designer, the IDE provides very different feedback (see Figure2(b)). Here we still see some parsing or classification problems but also data-flow errors, a dangling use-case reference, and multiple updates of an item within one use case (an issue for our test-planning engine). The issues of conditional statements and passive voice are not of concern here since they may have an interpretation in our test planner. Likewise, the unclassified action is reported as an "info" because the test designer may understand the implications for test-planning purposes.

Because we want to support different kinds of users/stakeholders in different development scenarios, our prototype IDE deliberately does not impose any intrinsic notion of use case consistency or completeness. Thus, it does not require the evaluation or enforcement of any particular



```
2 Actors, 2 Business Items, 9 Statements
The customer navigates to a shopping page.
The System displays a shopping page, including a catalog of CDs.
The customer enters the required number of each CD  (i.e. changes the numbe
The order is correspondingly updated.
Once happy with the quantities, the customer creates an order in the System
The system rejects an order and displays an error message if the order cont
The system determines a warehouse that can supply each line item.
If a warehouse is not found, the system returns the order back to the custo
System then initiates payment collection process using the customer's pre-r
If payment collection process is not successful, the system displays an err
Otherwise, it updates the warehouse for each line item in the customer's or
The system asks the warehouse to send the items to the customer's address.
```

```
3 errors, 2 warnings, 1 other
Errors (3 items)
    Dangling use-case reference:  initiated use case is null
    Use Case Statement is incomplete:  missing actor(s)
    Use error (item not created) :  Item 'order' used [UC0:3] when not created on path = UC0:0-UC0:
Warnings (2 items)
    Use case statement has more than one action (number of actions = 2)
    Voice of action 'is correspondingly updated' is not active (is PASSIVE)
Infos (1 item)
    Action 'is correspondingly updated' lacks an initiating actor
```

Figure 1. The Online Analysis Environment

predicate on use cases. The predicates that are available in a particular installation of the IDE are introduced via Eclipse extension points and so are a configurable element of the runtime environment. The predicates are grouped into suites, which can be tailored to particular user roles or development objectives. The evaluation of a predicate suite as a whole can be turned on and off. Within a suite, the severity level of a each predicate can be set individually (or even ignored). [1]

B. Implementation and Performance

Text2Test is implemented 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. In our experience the performance of the prototype is adequate to support regular, interactive development of use cases by iterative refinement. As an example, we analyzed a set of 16 industrial use cases (average length 8 sentences) for 28 inter and intra use case predicates [2]. The analysis found an average of 10 errors per use case. To complete a full analysis the IDE took an average of 15 seconds.

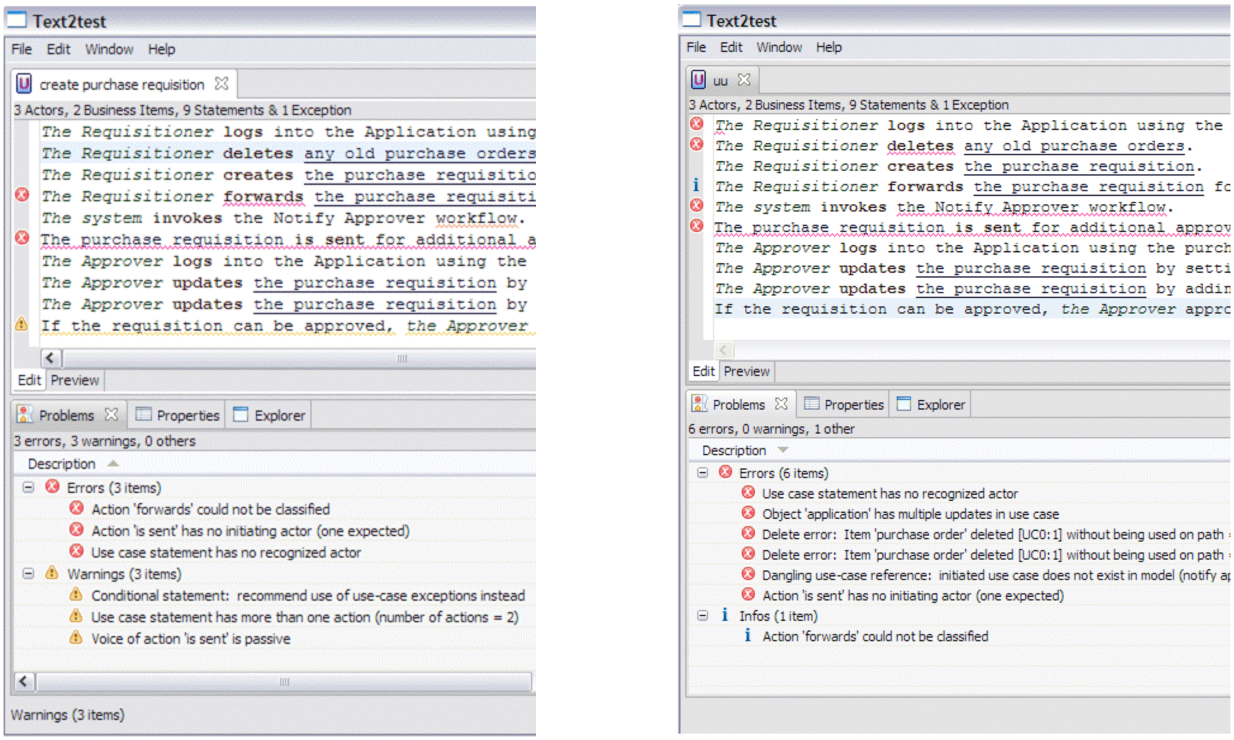
V. Model Analysis

We perform a linguistic analysis[3] of the use-case text to construct the model, then analyze the model consistency and other properties. Our text analysis technology is described in our previous publication [4]. Figure 3 depicts our metamodel. The information in a use case text can be summed up at the most abstract level in an application model. An application model contains a model of the context and a model of use cases. The context model contains a description of kinds of actors and business items. Each use case in the use case model contain a list of sentences. The sentences, in turn, contain one or more actions initiated by some actor of a kind described in the context model. An action may also have a set of parameters and can be associated to actors (also of a kind described in the context model) who receive the effect of the action. In addition, each action has a type and a voice (active/passive), while each parameter have one or more roles. While the type of action determines its classification, the role determines the relation of the parameter with respect to the action. Currently, the permitted action types are: Input, the act of providing information; Output, the act of receiving information; Create, the act of creating an entity; Query, the act of searching for an existing entity;

[1] We recognize that there are issues relating to the control of enforcement across the life cycle. These will generally involve aspects of management and process support that are beyond the scope of our current work.

[2] Hardware configuration: ThinkPad T61P, Windows XP version 2.11, Intel(R) Core(TM)) Duo CPU T7700 @2.40GHz, 790 MHz, 3.00GB RAM

[3] Linguistic analysis should be seen here as an enhancer and not the primary enabler. The models can be constructed via graphical editors, XSLT transformations and a myriad of other techniques supprted by the Eclipse modeling framework.



```
3 Actors, 2 Business Items, 9 Statements & 1 Exception
The Requisitioner logs into the Application using
The Requisitioner deletes any old purchase orders
The Requisitioner creates the purchase requisitio
The Requisitioner forwards the purchase requisiti
The system invokes the Notify Approver workflow.
The purchase requisition is sent for additional a
The Approver logs into the Application using the
The Approver updates the purchase requisition by
The Approver updates the purchase requisition by
If the requisition can be approved, the Approver

3 errors, 3 warnings, 0 others
Errors (3 items)
    Action 'forwards' could not be classified
    Action 'is sent' has no initiating actor (one expected)
    Use case statement has no recognized actor
Warnings (3 items)
    Conditional statement: recommend use of use-case exceptions instead
    Use case statement has more than one action (number of actions = 2)
    Voice of action 'is sent' is passive

Warnings (3 items)
```

(a) Feedback to a Business Analyst

```
3 Actors, 2 Business Items, 9 Statements & 1 Exception
The Requisitioner logs into the Application using the
The Requisitioner deletes any old purchase orders.
The Requisitioner creates the purchase requisition.
The Requisitioner forwards the purchase requisition fo
The system invokes the Notify Approver workflow.
The purchase requisition is sent for additional approv
The Approver logs into the Application using the purch
The Approver updates the purchase requisition by setti
The Approver updates the purchase requisition by addin
If the requisition can be approved, the Approver appro

6 errors, 0 warnings, 1 other
Errors (6 items)
    Use case statement has no recognized actor
    Object 'application' has multiple updates in use case
    Delete error: Item 'purchase order' deleted [UC0:1] without being used on path
    Delete error: Item 'purchase order' deleted [UC0:1] without being used on path
    Dangling use-case reference: initiated use case does not exist in model (notify ap
    Action 'is sent' has no initiating actor (one expected)
Infos (1 item)
    Action 'forwards' could not be classified
```

(b) Feedback to a Test Designer

Figure 2. Stakeholder specific error feedback

UPDATE, the act of updating the state of an entity; DELETE, the act of deleting an entity; DIRECT, the act of directing another actor to perform an action; INITIATE, the act of initiating another use cases; ACCESSCHANGE, the act of navigating between interfaces and UNCLASSIFIED. Roles of a parameter can be : ARGUMENT, the main argument of an action; HELPER, the parameter helping an action; PURPOSE, the objective of an action; etc. Some parameters can be linked to one of the business items defined in the context model. A statement in use case action can also be linked to zero or more exceptions. Each exception is guarded by a condition and contains a list of actions that must replace actions in the parent statement if the guarding condition is true (at runtime). Notice how the model preserves the syntactic and the linguistic aspects (sentence boudaries, action voices, etc.) along with the functional aspects of the use case (sequence of pre-classified and parametrized functions). We recognize that some downstream applications may require additional information than what is captured currently. The current meta-model is implemented using the Eclipse Modeling Framework (EMF) and is amenable to extension.

As an example, consider a snippet of a NL use case “Withdraw Money” : *The customer enters the withdrawal amount. The ATM returns cash to the customer and changes his account balance. If the balance on customer’s account is less than the withdrawal amount, the ATM returns an error.* The application model corresponding to this text is shown in Figure 4.

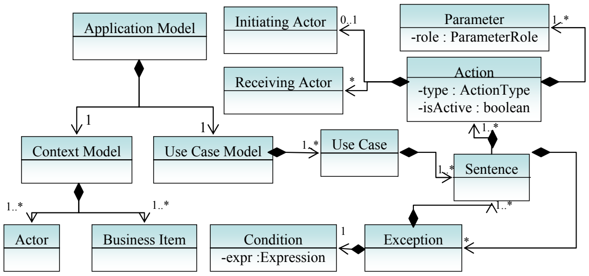


Figure 3. Use Case Description Metamodel

A. Reports and Predicates

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 summarize a model or describe

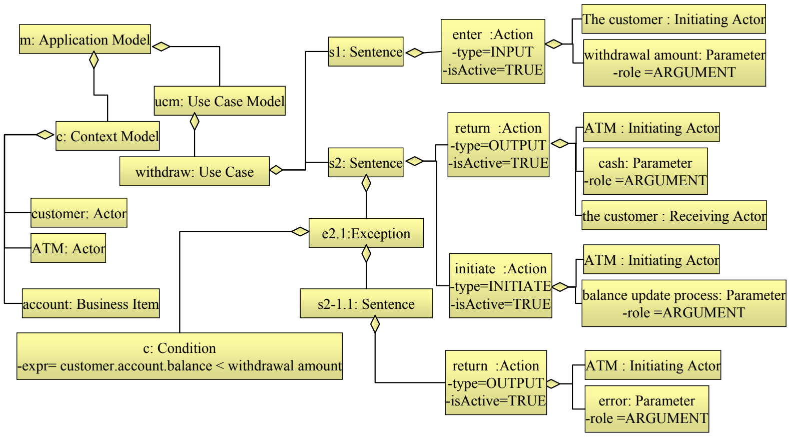


Figure 4. Instance of a Model

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 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 actions in a statement, the number of statements 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 control flow e.g., attempts to *use* items before they are *created*.

Concurrency-related checks, e.g., for possibly concurrent actions or possibly non-serializable behaviors

Inter-model checks to compare the actors and items referenced in a use case to 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 elements (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 environment, processes, products, and overall context of development.

B. Model Analysis Algorithms

In this section we give examples of algorithms that we use for model analysis.

1) Execution Paths: The ability to identify execution paths through one or more use cases is needed both for evaluating certain categories of predicates (such as for data-flow anomalies and serializability conflicts) and for generating test cases. Our analysis to identify sequential execution paths makes a few basic assumptions about control flow through the use cases:

1) Statements are executed in their given order
2) Branching of control occurs at conditional statements, with branches representing exceptions to the basic flow of the use case; if not explicitly stated exceptions are assumed to terminate without returning
3) A statement in one use case may invoke ("include") the execution of another use case, in which case the control flow will cross use-case boundaries; upon the termination of an invoked use case, control is assumed to return to the invoking use case
4) All flows that start from a single use case are sequential, with the exception that a single statement

may concurrently invoke two or more use cases; those use cases are considered to have potentially parallel execution
5) Number of iterations for a loop is pre-determined (thus, in effect, flattened)

These assumptions are shared and supported across the text analysis, use-case meta-model, and model analysis. Many are motivated by our prior work in test generation. The model-analysis engine supports the computation of all sequential paths (including cross-model paths) for individual use cases. It can be applied iteratively to compute all of the sequential paths for the set of use cases in a project. Concurrency issues can be evaluated for a set of sequential use cases (or use-case paths) by considering that they execute in parallel. Iterative effects can be considered by assuming the repetition of paths or path segments.

The main aspects of the use-case meta-model that are relevant for flow computation are that

- A use case consists (mainly) of a sequence of statements
- A statement consists (mainly) of a sequence of actions
- Some statements are conditional; these are represented in the model as exception statements that are associated to the nearest preceding non-conditional statement
- Some actions invoke one or more use cases
- Exceptions are semantically distinguishable into normal and abnormal flows (currently a dictionary based heuristic in text analysis).

An execution path is represented as a sequence of path elements, where a path element typically represents an action within a statement within a use cases–for example, the second action of the third statement of the eighth use case in the project. Special path elements mark exceptions or use-case invocations.[4]

Given the above, the basic logic for computing execution paths for the use cases in a project has the following outline:

```
Compute execution paths for use cases in a project:
{ For each use case in the project {
    Get the sequence of statements;
    Compute execution paths for the sequence;
  }
}

Compute execution paths for a sequence of statements:
{ For each statement in the sequence {
   For each action in the statement {
     If (the action is a "regular'' action)
        Add a corresponding element to the current path;
     If (the action is an "invocation'') {
       For each invoked use case {
         Add an "invocation'' element to the current path;
         Recursively continue to compute the current path
         with the statements of the invoked use case;
         Resume computation at this spot;
        }
      }
    }
    For each exception following the statement {
      Create a copy of the current path;
      Add an ''exception'' element to the path copy;
      Recursively continue to compute the copy path
      with the statements of the exception;
      If(exception denotes a ''normal'' flow){
            Terminate the copy path when the end of
               the exception statements is reached;
            Terminate the computation of that path;
       }
       else{
           Continue adding to the copy path after the
               exception-throwing use case statement;
        }
    }
  }
}
```

Note that this logic does not explicitly account for concurrency among invoked use cases; rather, possibly concurrent use cases are incorporated into the path in a sequence. This is a deliberate simplification. Our main interest in computing execution paths is in evaluating preconditions for test-case generation and then in generating the test cases themselves. The test cases themselves require sequential paths in which concurrent actions have been serialized. The preconditions for test generation thus require that concurrent actions be serializable, so in generating paths for test cases we assume that they are. Of course, in general concurent actions may not be serializable, but we test for the possibility of concurrency and non-serializable actions separately (discussed below). Only if those conditions are satisfied do we consider the path as suitable for test-case generation. Finally, invoked use cases may be executed in different serial orders. To avoid generating too many test cases we pick one particular order to generate initially. The logic above can be adapted to generate permutations of serial orders if necessary.

2) Data Flow Anomalies: We have defined a number of predicates that address data flow anomalies. We consider the identified business items as the "data." We test for anomalies such as references to items before they are created, deletions of items when they have not been used, and writes of items when they have not been read (among others).

Given a specific action on an item (that is, a particular action in a particular statement in a particular use case), and given a set of execution paths that include that action, it is straightforward to recognize an occurrence of a particular data flow anomaly. Suppose that we are interested in delete-delete anomalies, that is, the occurrence on a path of two operations that delete an item with no intervening operation that creates the item. That can be evaluated by the following logic:

```
Report delete-delete anomalies for a given action and
execution path set:

If (the given action is not a delete action)
  End;

For each path in the given path set
  For each occurrence of the given action on the path
    For each subsequent action on the path {
      If (the action does not refer to
```

[4] The information recorded in the path elements is sufficient to link back to the corresponding model elements, which in turn allow linking back to the original text.

```
      the same item as the given action)
    {
      Continue For each subsequent action;
    }
    If (the action is a delete action) {
      Report that the anomaly occurs;
      Continue For each occurrence ...;
    }
    If (the action is a create action)
      Continue For each occurrence ...;
  }
```

Directly analogous logic can be used to identify some other anomalies, such as consecutive creates without an intervening use or delete. Generally similar logic can be used to identify still other anomalies, such as a create not followed by a use.[5]

3) Concurrency Conflicts: Concurrency conflicts, such as non-serializable access to data, affect the quality of use cases and the artifacts that are based on them. We would like to identify non-serializable behaviors in use cases by static analysis (rather than by testing), and we would like to generate test cases, based on use cases, that do not suffer from problems such as read-write and write-write conflicts.

In our model the possible concurrent execution of use cases is directly represented by actions that invoke two or more use cases. On this basis we can identify and report possible concurrent behaviors. Beyond that, we can compute the set of execution paths for each invoked use case. Given these sets, we can perform comparative scans of paths from pairs of invoked use cases, looking for possibly conflicting uses of a particular business item.

VI. Evaluation

Research Questions: In this section we report results of an investigation that was carried out to evaluate the following:

1) Does a platform like Text2Test (one that enables edit time monitoring) ensure stronger compliance of use cases to a set of pre-defined guidelines?
2) Does a platform like Text2Test improve productivity of use case authors?
3) Does guideline-compliance of use cases ensure higher quality test cases in model based test generation?

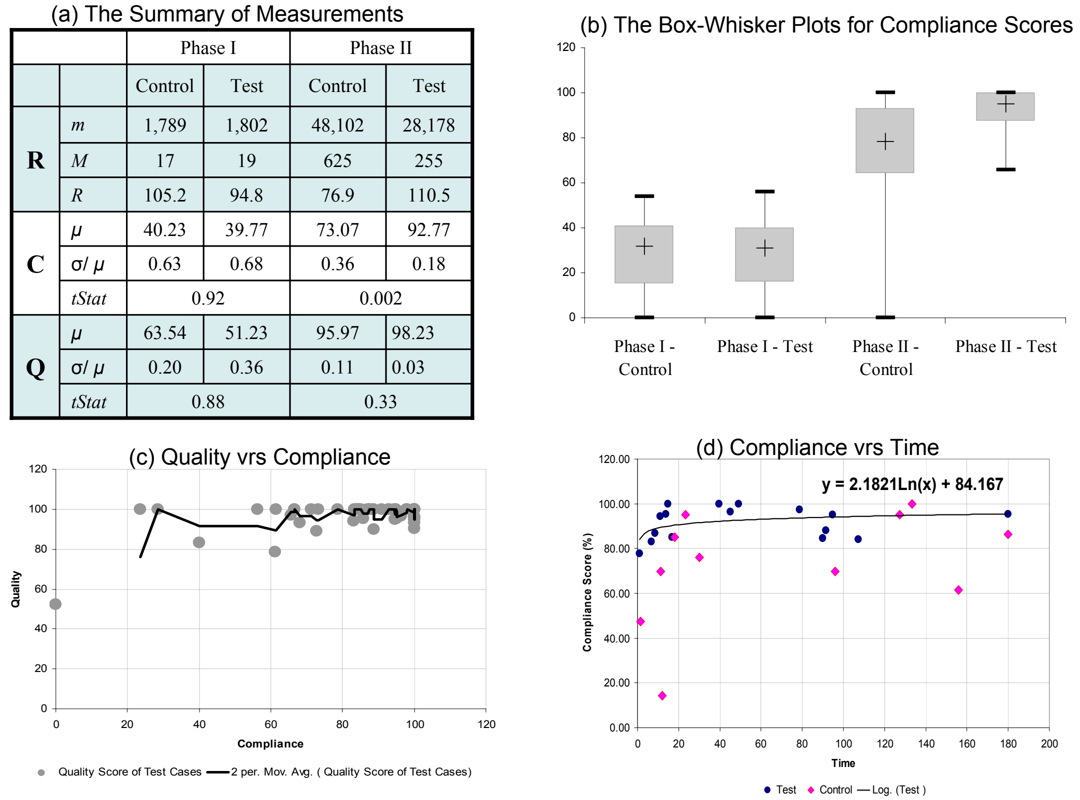
Experiment Process: The experiment was conducted *in-vivo*, but a controlled experiment was simulated. The use cases were collected from a live production environment in which ten subject matter experts (SMEs) were writing use cases, in plain English, describing the lowest level processes of a standardized business process hierarchy. The SMEs were experts in the domain of packaged application but had limited experience with use case authoring .The experiment was run for more than six months and consisted of two distinct phases: in *Phase I*, the SMEs wrote use cases without any guideline; and, in *Phase II* the SMEs wrote use cases with knowledge of a standard set of use case authoring guidelines. During *Phase II* a few SMEs were provided with the Text2Test tool. Thus, in effect, we had two groups of SMEs in *Phase II*: the test group , which used Text2Test to author use cases and the control group, which used Microsoft Word$^{\text{TM}}$to do the same. 5 SMEs were randomly chosen to form the test group. In between the two phases all SMEs were trained for a set of guidelines that eventually formed the basis of compliance evaluation. The use cases were stored along with their history in an asset management tool. Two months after Text2Test was introduced to SMEs a survey was conducted to record their subjective assessments.

Metrics:Compliance of a use case to the guidelines was measured as $C = \frac{n}{N}$, where n is the number of non-compliant lines of use cases and N is the net number of lines in the use case. Compliance was automatically computed for each use case using a set of style and content predicates implemented on Text2Test. The same set of predicates were used to train the SMEs on use case authoring guidelines. To measure productivity we used the subjective assessments of the test group and also measured the rate at which use cases were checked into the repository. The rate was measured as: $R = \frac{m}{M}$, where m is the number of lines of use cases checked in to the repository by a group and M is the number of "Man-Days" for the group. The SMEs had a schedule focussed on authoring use cases. Therefore, we did not have to discount for "out-of-production" time. Also, only the final check-in of use cases were counted and intermediate check-ins were ignored. The quality of a test case generated from the use case was measured as $Q = \frac{t}{T}$, where t is the number of steps in the test case that had complete information and T is the net number of test steps in the test case. The test cases were generated automatically for the use cases using the extracted model and a strategy of *all-paths* through a *def-use* graph. A test step is deemed to have complete information if it contains information about its initiator, its arguments and its classification (as to execution or verification step). We realize that our quality assessment of test cases is subjective. However, since the project used the same test generation strategy (and tool) as ours, the quality assessment reflects on that of the project.

Results: Figure 5(a) summarizes our measurements.[6] The first row of measurements depicts our findings on the productivity metrics. The productivity metric R declined from Phase I to Phase II in the "Control" group; it shows the effect of introduction of a process without a tool support. In stark contrast, for the "Test" group, we find that there is a gain in productivity. This shows how a good tool support can enforce a process and may even enhance the productivity.

[5] We realize that there are numerous static analysis frameworks available, but the approach of encoding each data-flow predicate idividually is simple and adequate to our current needs.

[6] The complete set of measurements and the survey results can be obtained by e-mailing the authors.



(a) The Summary of Measurements

		Phase I		Phase II	
		Control	Test	Control	Test
R	*m*	1,789	1,802	48,102	28,178
	M	17	19	625	255
	R	105.2	94.8	76.9	110.5
C	*μ*	40.23	39.77	73.07	92.77
	σ/ *μ*	0.63	0.68	0.36	0.18
	tStat	0.92		0.002	
Q	*μ*	63.54	51.23	95.97	98.23
	σ/ *μ*	0.20	0.36	0.11	0.03
	tStat	0.88		0.33	

Figure 5. Summary of Results

During Phase II, the “Test” group has a productivity of 110.50 lines/ Man-day, which is much greater than that of the “Control” group: 76.96 lines/Man-day. The survey of the SMEs that was conducted two months after introduction of Text2Test, confirmed our productivity findings. According to one SME, “The use of Text2Test helped achieve productivity boost by constantly providing a feedback about compliance.” Another SME commended the non-interfering style of the feedbacks.

We noticed considerable improvement in “Compliance” and “Test Quality” from Phase I to Phase II. This shows that the introduction of guidelines were effective for both the “Test” and the “Control” group. However, the “Test” group has a significantly higher average compliance (a gain of 19.70%) than the “Control” group. This demonstrates how a proper tool support is necessary to support guidance enforcement. We can rate this finding as statistically significant because the “tStatistic” computed by performing a two tailed heterogeneous t-test has a value of 0.02 (implying only 2% chance that the “Test” and the “ Control” group measurements belong to same population). Figure 5(b) shows a box plot of the two groups during PhaseI & II. The “Test” and the “Control” group populations show significant differences in Phase II but they look similar during Phase I. This further validates the significance of our finding that proper tool support enhances guideline compliance.

The gain in compliance also results in higher quality of automatically generated test cases. In Figure 5(c) we depict a scatter plot between “Compliance” on X-axis and “Test quality” on Y-axis. Notice that they reflect a positive correlation: the Karl-Pearson Correlation Coefficient was measured at 0.72. The moving average fitted on the data shows how the test quality converges towards 100% with 100% compliance of use cases.

Figure 5(d) points to another interesting observation. When we plot the Phase II compliance scores against the time since the beginning of the phase we notice that for the test group there is a logarithmic growth. No trend is observed for the control group data. This confirms the intuition that a tool like Text2Test can provide maximum benefit to an un-experienced user. With time, the benefit becomes increasingly marginal. The control group compliance vrs time data was more erratic. Compliance in the “Control” group depended more on the personal style of the author and showed little change with time. This fact is further exemplified by the relatively huge standard error (measured as $\frac{\mu}{\sigma}$) in “Compliance” scores for the “Control” group in Phases I & II: 0.63 & 0.36 respectively (see Figure 5 (a)). Phase I “Test” group std. error ”Compliance” score, was similar to that of the “Control” group phase I. In phase

II, std. error on“Compliance” score for the “Test” group reduced significantly to 0.18. Low std. error implies lower variance and a general trend of improvement across the group. Similar, observations can be made in the “Quality” scores of the two groups in Phases I & II.

Threats to validity: The experiment design minimizes the effects of the threats to internal validity [15]. Biases resulting in differential selection of SMEs for the comparison groups are eliminated through blocking of the effect of the controlled variable, proficiency of SMEs, through randomization. The effect of instrumentation is minimized because the measurements are mostly automated and are performed by a single person at the end of the experiment. The SMEs were monitored through repository data on their daily performances. This gave us a chance to observe any effects due to History and Maturation. Post-mortem of the repository log gave us no indication of such effects. Some data was collected through surveys and self reporting of the data poses another threat to the experiment design. Since our measurements confirm our survey findings risk of self reporting is minimized. Concerning the external validity, the use of SMEs as subjects is a threat. Some authors of use cases may have less experience in their problem domain than our subject SMEs and such authors may commit more domain specific errors than compliance related errors. While this may affect the validity of productivity metrics, this is not a significant threat to the compliance metric.

VII. Summary, Conclusions, and Future Work

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

- Construction of abstract models of use cases that contain both syntactic and functional information in use case text.
- Automated analysis of models of use cases for a customizable and extensible variety of quality properties
- Support for write-analyze-revise cycle of use case refinement.

The approach is embodied in Text2Test, a prototype IDE built on Eclipse. As shown in an industrial setting, the use of Text2Test in writing the use cases can greatly improve guideline compliance, enhance productivity and increase the quality of automatically generated test cases.

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.

Acknowledgments

We are grateful to Mr. Palani Kumanan for his help with Text2Test. We thank Dr. 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 participation in the evaluation of Text2Test.

References

[1] I. Jacobson, “Object-oriented software engineering - a use case driven approach,” in *TOOLS (10)*, 1993, p. 333.

[2] C. Rolland and C. B. Achour, “Guiding the construction of textual use case specifications,” *Data Knowl. Eng.*, vol. 25, no. 1-2, pp. 125–160, 1998.

[3] G. Fliedl, C. Kop, H. C. Mayr, C. Winkler, G. Weber, and A. Salbrechter, “Semantic tagging and chunk-parsing in dynamic modeling,” in *NLDB*, 2004, pp. 421–426.

[4] A. Sinha, A. Paradkar, P. Kumanan, and B. Boguraev, “A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases,” in *DSN ’09*, 2009, pp. 327–336.

[5] A. Cockburn, *Writing Effective Use Cases*. Boston, MA, USA: Addison-Wesley, 2000.

[6] C. Karl, A. Aurum, and R. Jeffrey, “An experiment in inspecting the quality of use case descriptions,” *Journal of Research and Practice in Information Technology*, vol. 36, no. 4, 2004.

[7] B. C. D. Anda, D. I. K. Sjøberg, and M. Jørgensen, “Quality and understandability of use case models,” in *ECOOP 2001*, 2001, pp. 402–428.

[8] D. Jagielska, P. Wernick, M. Wood, and S. Bennett, “How natural is natural language?: how well do computer science students write use cases?” in *OOPSLA ’06*, 2006, pp. 914–924.

[9] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, “Application of linguistic techniques for use case analysis,” *Requirements Engineering Journal*, vol. 8, no. 3, pp. 161–170, 2003.

[10] ”RavenFlow Inc.”, “”www.ravenflow.com”,” 2008.

[11] F. Törner, M. Ivarsson, F. Pettersson, and P. Öhman, “Defects in automotive use cases,” in *Int. Symposium on Empirical Soft. Eng. ’06*, 2006, pp. 115–123.

[12] A. Sinha, M. Kaplan, A. Paradkar, and C. Williams, “Requirements modeling and validation using bi-layer use case descriptions,” in *MoDELS ’08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 97–112.

[13] M. Kaplan, T. Klinger, A. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, “Less is more: A minimalistic approach to uml model-based conformance test generation,” in *ICST ’08*, 2008, pp. 82–91.

[14] OMG, “Business process modeling notation version 1.1,” *http://www.bpmn.org/Documents/BPMN1-1Specification.pdf*, 2008, object Management Group.

[15] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Company, June 1966.