Automating Requirement Analysis Through

Combination of Static and Dynamic Model Construction

Problem Statement

Most requirement specifications are written in natural language. This approach facilitates easy communication among stakeholders because everyone can understand clearly written natural language text. However, the drawback is that the natural language specifications are often imprecise and ambiguous. There have been many proposed ways to tackle this problem. Formal modeling, controlled grammar and natural language parsing are the most widely known approaches. The Dowser project aims to create a static model of a system using link grammar parsing of requirement specifications. Project Cogito uses a controlled grammar to parse use-cases and validates the use-cases against domain specific knowledge stored in an ontology. We are trying to combine these two approaches to generate static and dynamic model and explore the relation between them. We will implement a unified system using the Dowser and Cogito. This system will provide a more comprehensive view of the system than that produced by any single tool itself.

Background

Requirements specification can be enhanced in several ways.

1. Learning to write unambiguous and precise specifications
2. Detecting ambiguities and inconsistency in the specification
3. Using a controlled language to restrict the specification text

Project Dowser:

Project Dowser uses the second approach. Link grammar parsing is used to extract static OO constructs from the requirement specification. It uses syntactical knowledge to build an object oriented analysis model. Dowser applies syntactical rules on the created dependencies produced by link grammar parser. These rules define how to extract classes, associations and methods. Every rule corresponds to a UML diagram template which is then filled with the actual text from the specification. The final result is a UML class diagram. Since the produced diagram is smaller than the original specification and clearly visualized, it enhances understanding, and a human can check it for ambiguities. Any defects can be traced back to the original specification. Dowser provides high recall but it does not guarantee high precision because the generated static model has to be assessed again by human experts.

Project Cogito:

Project Cogito uses the 3rd approach. It consists of two tools viz. 'Use-Case Editor' (UCE) and 'Ontology Development Tool' (ODT). To tackle the problem of ambiguity in natural language specification, a controlled language is developed for writing use-cases. The UCE tool provides an easy-to-use interface which shows context specific suggestions while writing use cases. The domain knowledge pertaining to a particular application area is stored in an ontology. This ontology can be developed using ODT. UCE tool uses the stored ontologies to validate use-cases.

Integration of Dowser and Cogito:

Since these two approaches provide static and dynamic models respectively, we are trying to combine the two techniques to evaluate the effectiveness of the integrated tool. For combining the two techniques we are porting both tools onto Ubuntu. Cogito makes use of the Java Swing API. A unified Java GUI is provided for the whole system. Using this combination of static and dynamic model we provide a more comprehensive view of the system to the user. The user can also tweak the models through a GUI.

Related work

There have been several attempts to generate a system model from various types of requirement specification documents. Most of those can be categorized at different levels as follows:

1. Defining a formal way / guideline on specifying the requirements and re-writing specifications accordingly.
2. Analyzing currently written documents to detect ambiguities / inconsistencies and asking users take corrective measures.
3. Automatically suggesting unambiguous and consistent usage of terms in requirement specifications.
4. Analyzing the requirement specifications and, after converting it into some intermediate language, generating code from the intermediate language.

Many have suggested use of domain knowledge in some form to assist the process of disambiguation. Several ways of representing knowledge such as OWL, RDF, SBVR, DRS, simple XML etc. have been proposed. Raj et al. [(1)] have developed a technique for transformation from SBVR (Semantics of Business Vocabulary and Rules) to UML models. GATE (General Architecture for Text Engineering) by H. Cunningham [(2)] uses TIPSER data model for storing knowledge. Non-standard representations such as DRS (Domain representation structures) in Fuchs's ACE (Attempto Controlled English) [(3)] system are also used. Mauco et al. [(4)] try to bridge the gap between natural language and formal description by choosing LEL as natural language input and

RAISE as formal language output. But the LEL specification itself is hard to write and hence requires quite a lot of effort from the analyst. Grieskamp et al. [5] have developed an abstract state machine language called ASML, which is an executable specification language. Use-cases can be encoded into ASML using the authors' mapping scheme. They also give test-generation and test-oracle algorithms. The tool is now fully incorporated into Visual studio. While ASML has all these advantages, it doesn't use a static model for validating the system structure. Bensuk et al. [6] suggested automation of the process of converting requirements to design using a two-level grammar (TLG) as a bridge between natural language requirements and formal specifications. Although the implemented Contextual Natural Language Processing technique seems to perform better, it is still limited by the part-of-speech tagger and the corpora used for resolving syntactic ambiguity. It does not use any generic dictionary, and it also does not provide any extension mechanism.

Using generic information extraction framework for software engineering tasks [2] is shown to be effective. However, if we are applying requirements engineering specific techniques such as combining use-cases with static models, we will benefit from the fact that this solution will be tailored to the requirements analysis process.

There have been some approaches for synthesizing simple and hierarchical state machines directly from requirements [7] [8]. But these approaches are complicated, limited to particular type of requirement texts and also do not have a way to extend the model of the system. Providing different views of the system to avoid ambiguity and ensure completeness is a proven approach to requirements engineering. We are trying to provide sequence diagrams (easily derived from use-cases) with static diagrams (constructed by Dowser). This approach will capture the dynamic as well as static behavior of the system.

Our approach is very close to the controlled grammar technique used in ACE [3]. However, ACE does not use link grammar parser or any dictionary such as WordNet. Hence it supports limited uses of nouns and verbs. We wish to create a grammar which will be more expressive than ACE and also will support domain specific terms taken from an ontology.

We have written some checks to be performed individually on both static and dynamic models. For cross-model verification, we studied techniques from [9] and [10]. We also use word senses and relations provided by the WordNet database libraries.

Earlier Version

Dowser:

- Uses link-grammar parser and post-processing rules
- Runs on Windows using Cygwin libraries
- Requires installation of the WordNet-3.0 and Graphviz packages
- Uses UMLGraph for generating static diagrams
- Written in C++ programming language

Cogito:

- Uses ANTLR-generated lexer and parser
- Runs on Windows using the .NET platform
- Stores use-cases in serialized .NET object format
- Written in C# programming language

Integration plan

Dowser:

- Dowser was ported completely on to Ubuntu by removing the Cygwin dependencies.
- Dowser's Printer component was modified to create a simple object model stored in a flat file.
- A simple parser was used to create in-memory representation of the object model.

Cogito:

- Cogito was ported completely on to Ubuntu by rewriting it using Java Swing API.
- Intermediate XML representation for use-cases was defined:
  - After parsing the use-cases, the generated AST is stored in addition to the use-case itself.
  - Use-cases are stored using the pre-defined XML schema.

Combined approach:

- A Java GUI application acts as a single entry point to the whole system.
- The application provides menus to invoke Dowser, the Use-case editor, and the Ontology editor components.
- Static (class diagram) and dynamic (sequence diagram) views are shown in Dowser and Use-case editor UI respectively.

System Architecture:

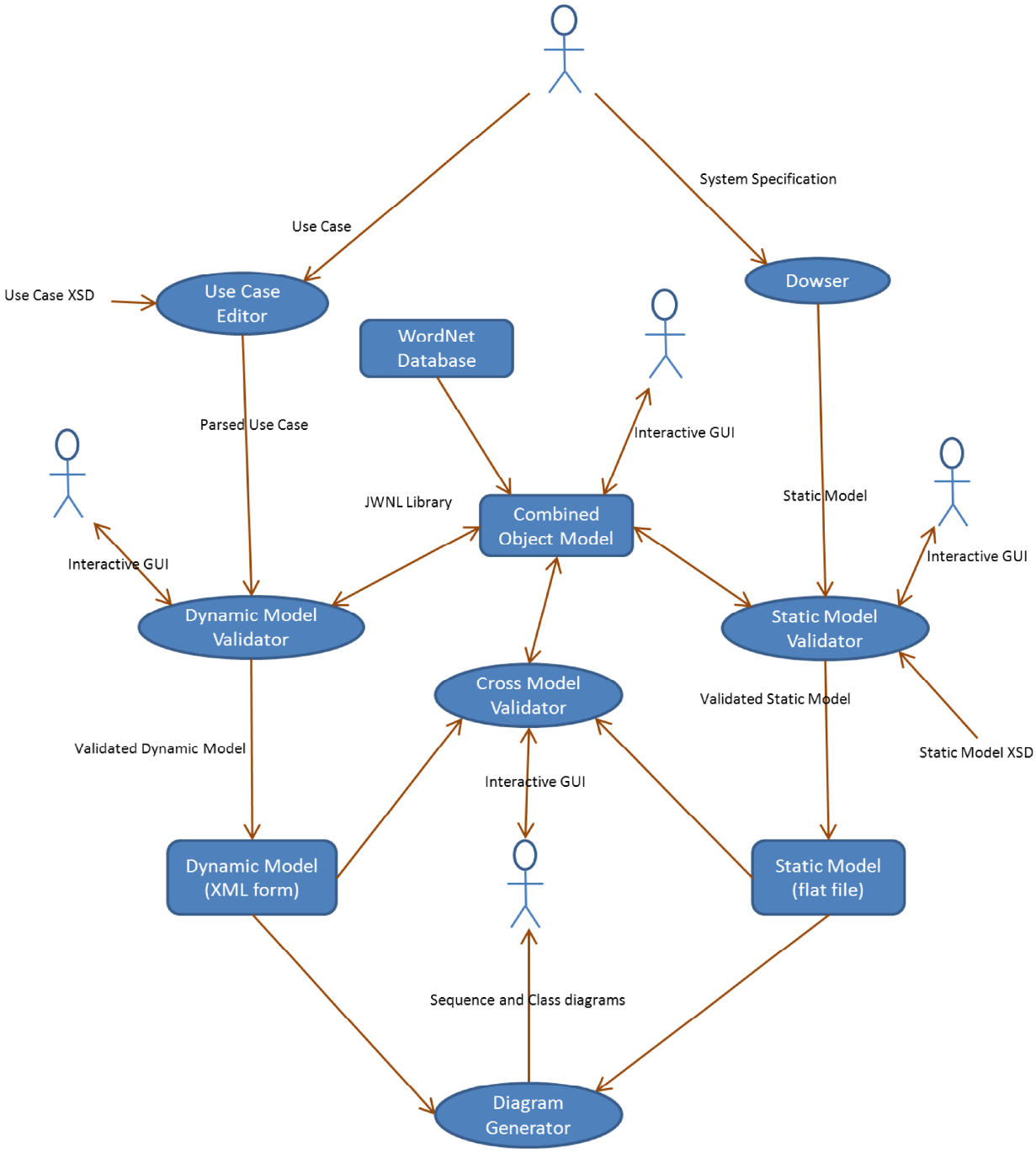


Figure 1

Architecture Description:

Figure 1 shows the overall system architecture for the combined Cogito and Dowser system. The user provides two forms of input; textual system description and a set of use-cases to Dowser and Cogito respectively. The two components then generate the static and dynamic models of the system, which are validated by the respective validators. The user can see the generated diagrams along with the model (in textual format). He can then tweak the model according to his needs. The validated models can then be cross-checked by the system to provide appropriate suggestions.

Component Description

Dowser:

The Dowser GUI component (shown in figure 4) facilitates writing of system specification. It invokes the dowser classExtractor to extract a static model from the specification. The static model will be presented in a UML class diagram using UMLGraph library. The static model is also stored in a flat file. The file can then be parsed to create the in-memory model used for validating use-case.

Static Model validator:

The static model validator component will validate the static model of the system using the ontology. I will use the dictionary interface wrapper based on WordNet library. It will also allow user to tweak the validation process.

Checks that are performed during this process are:

- Each class has a noun form
- Each attribute has properly specified type
- Each relationship is represented by either an adjective or action verb
- In an inheritance relationship, superclass is identified as a hypernym of the subclass OR subclass is identified as a hyponym of the superclass
- In an aggregation relationship, composite class is identified as a meronym of the part-class OR part-class is identified as a holonym of the composite class.

Use-Case Editor:

Th use-case editor (written in Java Swing) (shown in figure 5) provides an interface for the user to write use-cases. It also provides on-the-fly suggestions while writing the use-cases. After completing the use case writing, the user can choose to validate the use-case. During the validation, various checks are performed on the use-case. These checks include the validation of concepts and actions used in the use-case against the static model constructed from the

system specification. A dictionary interface that wraps the WordNet library is used to validate the used terms. The pre-defined XML schema is used to store the use-case.

Dynamic Model Validator:

The dynamic model validator validates the dynamic behavior of the system using the ontology. The use-case is validated against the static model as well as WordNet library. The user can tweak the model during validation process.

Checks that are performed during this process are:

- Each actor in use-case is a noun
- Each action taken by an actor (subject) is a valid verb

Ontology Editor:

The ontology editor (shown in figure 2) provides a GUI similar to Dowser GUI. User can add and modify classes, relations, attributes, operations etc. The domain ontology stored in the ontology model is combined with the auto-generated static model to validate use-cases. The ontology model is also used while validating the static model.

Cross-Model Validator:

The cross-model validator validates the models generated by Dowser and use-case editor. The following cross-model checks are applied:

- Each actor in use-case is represented by one and only one class in the static model
- Each action taken by an actor (subject) is a valid method on the subject's class
- Each pre / post condition involves some attribute of a class
- Each attribute takes only allowed values
- Relationships between classes are either realized by attribute or action
- Every adjective in a use-case other than the class's own attributes should be expressed by some relationship with another class in the static diagram
- Every action in a use-case should be either the class's own method or it should be expressed by some relationship with another class in the static diagram

Diagram Generator:

The ‘UMLGraph’ library is used for creating static and dynamic diagrams. For generating static diagrams, the tool creates annotated java files that are given as input to ‘UMLGraph’ library which in turn creates a ‘dot’ file for the ‘graphviz’ tool. This dot file is then converted to an image using the ‘graphviz’ library. For generating dynamic diagrams, ‘UMLGraph’ library needs ‘pic2plot’ program which is available in the ‘plotutils’ package on Ubuntu. Input to the pic2plot program is a macro file provided with the ‘UMLgraph’ distribution and a file which includes

information about drawing lifelines and messages. ‘pic2plot’ program outputs a postscript (ps) file. For displaying this image we use image conversion program ‘convert’ from the ‘imagemagick’ package on Ubuntu.

Combining the static and dynamic model of a system:

Reason:

Dowser and Cogito present different views of a system using static and use-case models respectively. These approaches alone are not enough to produce a comprehensive model of the system. Also any dynamic model of the system cannot exist on its own. It needs a basic static model in order to add behavioral aspects.

Benefits:

Static and dynamic model complement each other to provide a complete view of the system. This helps us move one step further towards generating a comprehensive model of the system. This systematized specification also ensures that the validation process will be smooth and less error-prone. We will try to use Dowser’s approach to build static model and the ontology that is an input to the Use-Case validator component. This combination also facilitates cross-model consistency verification.

Issues faced:

- Dowser uses a link grammar for extracting the static structure of a system. This is possible because a static model can be described with a comparatively small number of language constructs. However, dynamic model extraction using Dowser-like parsing techniques requires considerable amount of work. This is due to the fact that the behavioral facet of the system is generally specified using more grammatical constructs, and hence it becomes hard to specify and parse. We needed to either convert the large set of grammar rules used in Cogito or to create a framework which is agnostic to these parsing techniques but still has the ability to use the output generated from both the components.
- Dowser outputs a static diagram using a file model. This file is then used by ‘UMLGraph’ and ‘graphviz’ to generate the actual diagram image. We needed in-memory model of the static diagram hence we had to add an extra functionality into dowser to output the static model into a file which can be parsed into an in-memory object model.
- Cogito was written completely in .NET platform including Windows Forms GUI. Since we wanted to unify the two projects, we needed a common platform and technology to be used. We had to port Cogito to Ubuntu using Java Swing. Porting the parser component from cogito was easy since ANTLR can directly produce Java lexer / parser.

- Input source
  We had to define the form of the input that these two tools use. Dowser uses requirements specification which adheres to specific rules while Cogito restricts user from writing syntactically invalid constructs. In order to fully utilize the potential of the two separate tools, we did not change the input form but we created a unified interface for a user to access both the tools.

Tool Screenshots

Figure 2 : Dowser Main GUI

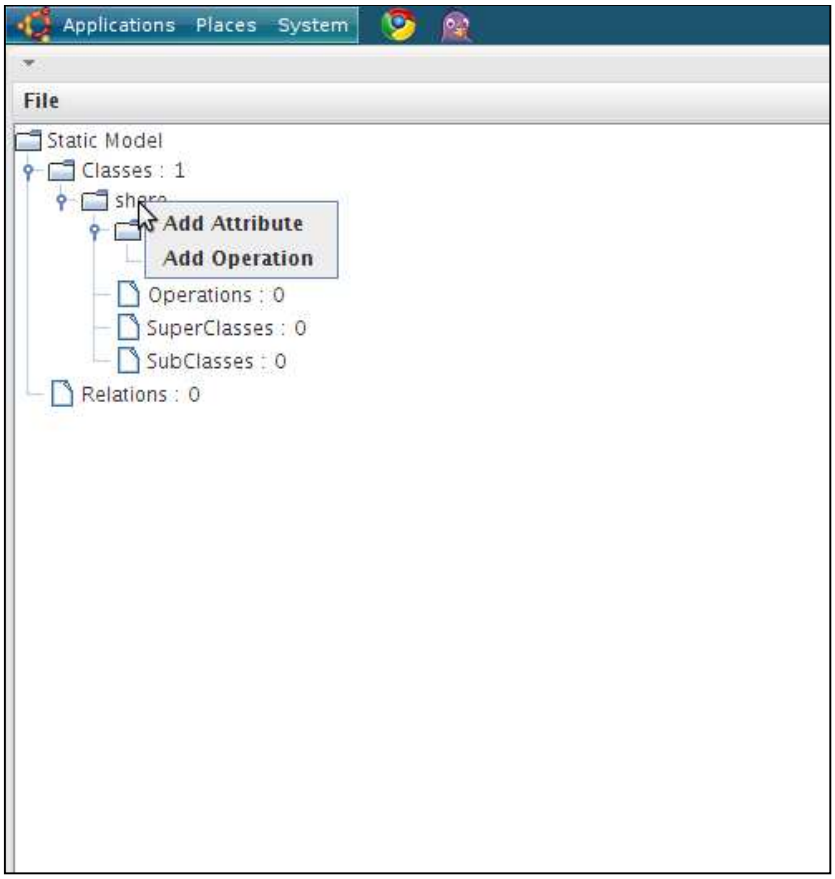


Figure 3 : Ontology Editor GUI

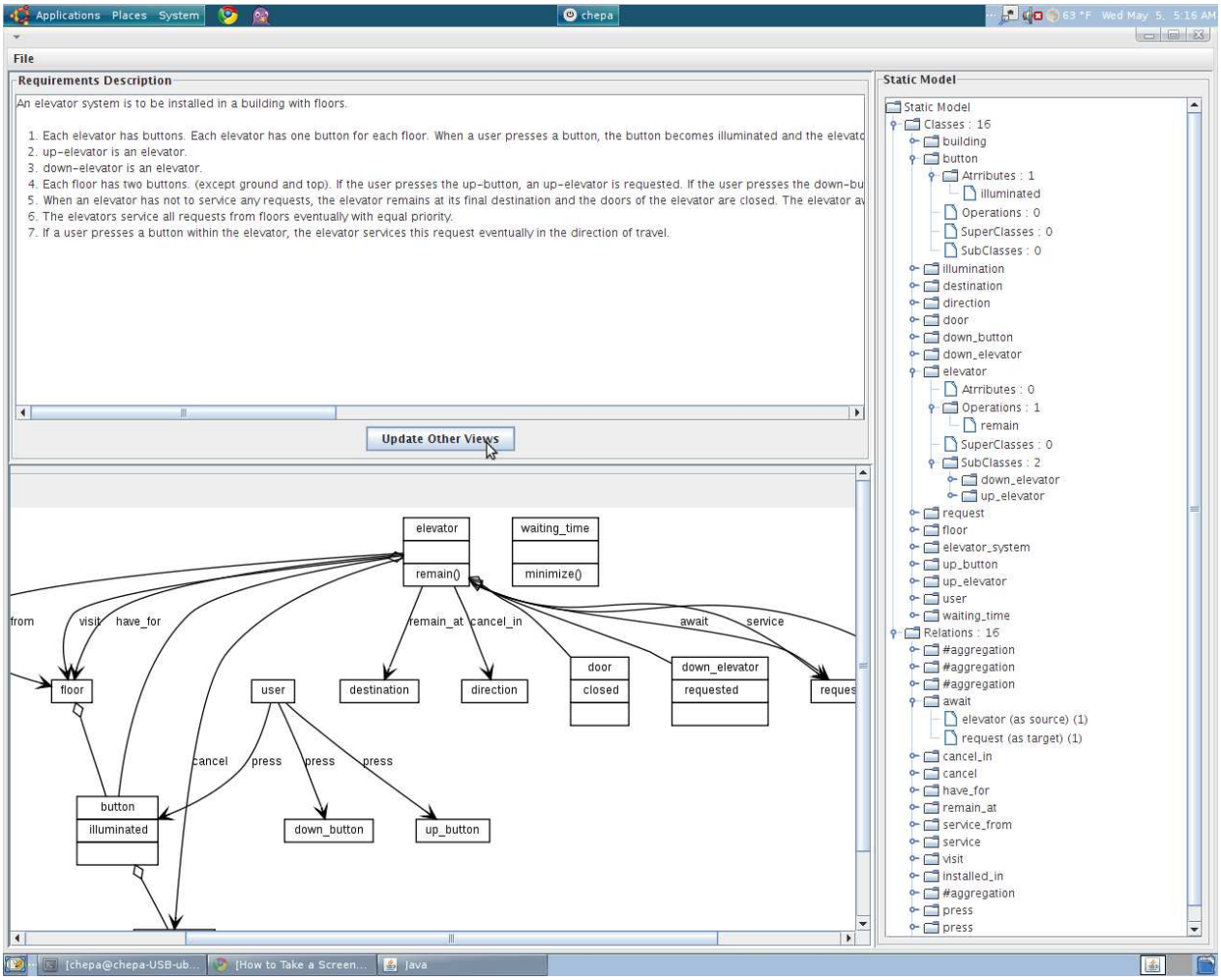


Figure 4 : Dowser GUI

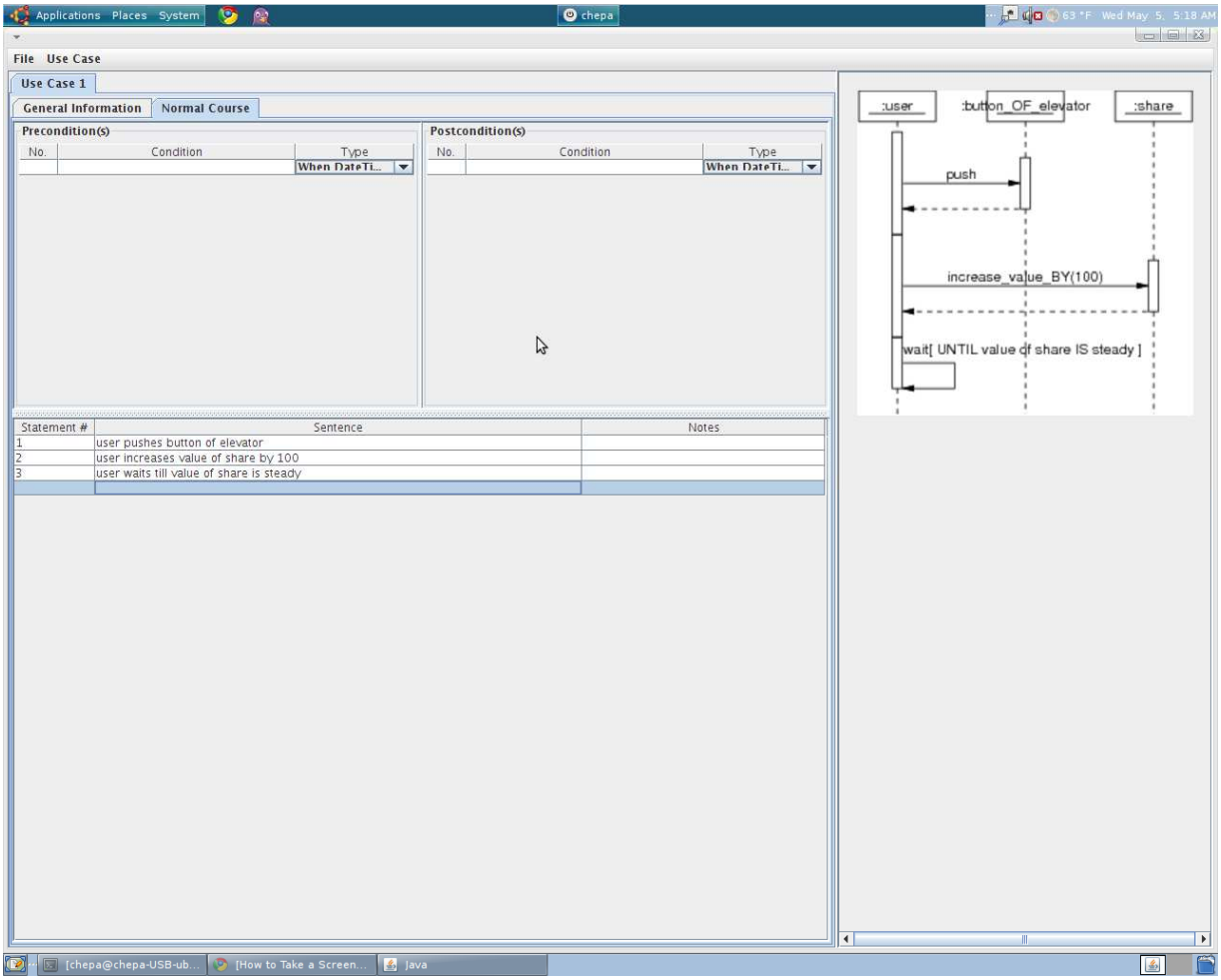


Figure 5 : Use-case Editor GUI

Case study:

We used following textual description for elevator system.

"An elevator system is to be installed in a building with floors. Each elevator has buttons. Each elevator has one button for each floor. When a user presses a button, the button becomes illuminated and the elevator visits the corresponding floor. When the elevator visits a floor, the elevator cancels the corresponding illumination. Up-elevator is an elevator. Down-elevator is an elevator. Each floor has two buttons. If the user presses the up-button, an up-elevator is requested. If the user presses the down-button, a down-elevator is requested. If the user presses a button, this button becomes illuminated. When the elevator visits a floor, the elevator cancels the corresponding illumination of the button in the desired direction. The system minimizes the waiting time. When an elevator has not to service any requests, the elevator remains at its final destination and the doors of the elevator are closed. The elevator awaits further requests. The elevators service all requests from floors eventually with equal priority. If a user presses a button within the elevator, the elevator services this request eventually in the direction of travel."

The Dowser GUI creates the following static diagram (figure 6).

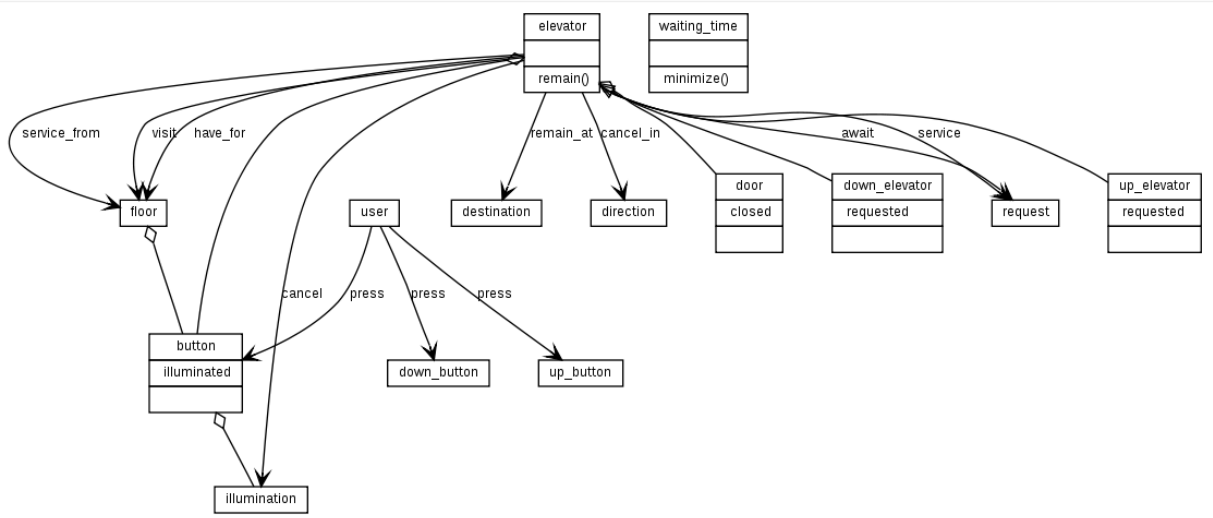


Figure 6 : Static Diagram

After applying the checks,

- The static model validator determines that 'up_button' is not an actual noun (using WordNet). However, there is a class called 'button' which could have some relationship with 'up_button'. The tool displays a message to indicate that user could specify some relationship between 'up_button' and 'button'.

- We can then modify the description. For example, we add an inheritance relationship between ‘button’ and ‘up_button’. The new static diagram will then be validated again.
- The elevator_system doesn’t have any relationship with elevator. The validator displays a suggestion indicating that we can add some relationship between elevator_system and elevator.

Now we consider use-cases written for the elevator system.

- Sentence : *“User pushes up_button of elevator”*

After parsing the sentence, the dynamic model validator checks whether a ‘push’ action is specified for up_button. Since there is no such action specified in the description, it displays a message indicating that we should re-write the sentence using actions which have similar meaning. In this case, ‘press’ is an equivalent action that can be used. In spite of the problem, we can still see the sequence diagram as shown in figure 7.

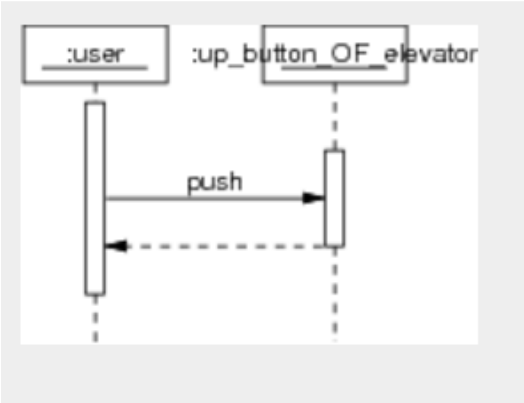


Figure 7

- Sentence : *“User waits for 20 secs”*

The validator indicates that there should be some action ‘wait’ associated with ‘user’ class. But since this action need not be a part of the model, we can choose to ignore this. The sequence diagram is shown in figure 8.

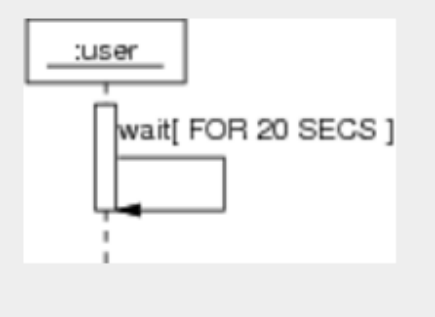


Figure 8

- Sentence : "*elevator_system illuminates button*"
  The validator checks whether illuminate is a valid action on 'button' class. Since such action is not provided a message is displayed indicating that some equivalent action should be specified in the description. In this case, there was an attribute 'illuminated' in the 'button' class. However the validator fails to take this into account.

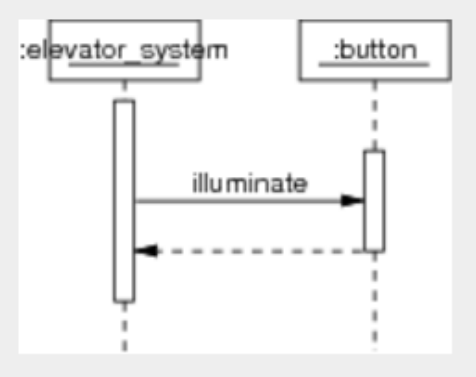
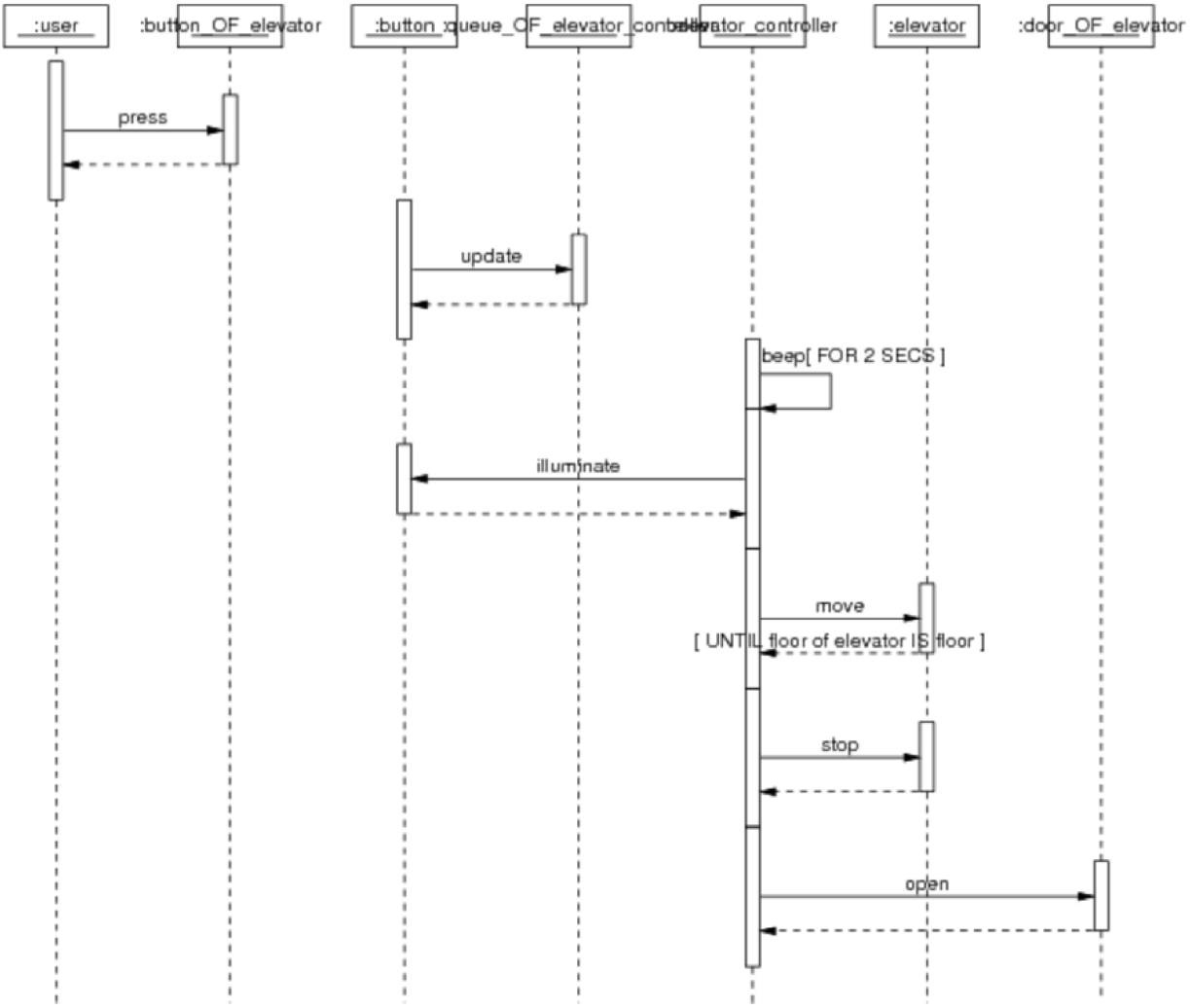


Figure 9

- Figure 9 shows the complete sequence diagram for the following use-case.
  - User presses button of elevator.
  - Button updates elevator_controller.
  - Elevator_controller beeps for 2 secs.
  - Elevator_controller illuminates button.
  - Elevator_controller illuminates button.
  - Elevator_controller moves elevator till floor of elevator is floor of user.
  - Elevator_controller stops elevator.
  - Elevator_controller opens door of elevator.



- Guard conditions are also represented on the message line.

Validation process

The validation process is able to determine inconsistencies between models for simple relationships such as inheritance and 'part-of'. It also detects guard conditions and shows them in the sequence diagrams accordingly. It also displays informative messages which could be used to modify the use-cases or the textual description or both. The suggestion process uses extensive knowledge stored in WordNet dictionaries. However, we need a more powerful parsing technique for use-cases which will enable us to gather more information from it.

Discussion and future work:

- Comparison with individual cogito or dowser
  Dowser and Cogito respectively provide static and dynamic views of the system. Although the static and dynamic views are easy to understand when considered alone, there remains inconsistency between the models if they are not cross-checked. Our solution addresses this problem and provides a more comprehensive view of the system.
- Combining grammars into a single set
  The parser technologies used in these two approaches are dissimilar. But since we are keeping the two components separate until the cross-model validation takes place we do not need to worry about this difference. We might explore the possibility of using the link grammar throughout the entire system. This will require conversion of Cogito grammar into link based grammar.
- Statistical parsers
  We studied techniques which use statistical parsing for analyzing natural language texts instead of context-free or link grammars. Although these techniques are widely used in natural language parsing, we haven't explored use of such parsers in the automation of requirements analysis process. A survey of such candidate statistical parsers could be conducted in the context of improving effectiveness of our system.
- Synthesizing state charts
  Although sequence diagrams provide a good view of the system's dynamic behavior, state-charts are more compact and also provide a different dynamic view of the system. The ability to synthesize state-charts from set of sequence diagrams would be the next logical step in this automation process.
- Ontology import
  The combined tool uses hand-written ontology and the static model generated by Dowser. Thus we have achieved semi-automation of knowledge-base generation. However, we can provide mechanism to allow imports from other ontology formats such as OWL. This will allow us to re-use a large size knowledge-base which is already written.
- Improvements in generating the graphical representation
  Currently, UMLGraph library is being used to generate static and sequence diagrams. This involves many transformations on the input to make it suitable for diagram generation. We also need many packages such as 'plotutils', 'imagemagick' to make the generated image suitable for display. We need to explore other possibilities of drawing UML diagrams from declarative specifications. Poseidon, ArgoUML are good candidates for this purpose.
- Case study on Industry level projects:
  To evaluate the technique more precisely, we need to exercise industry level requirements using the combined approach. Such a case study will involve extensive set of requirement

specifications and use-cases. It will enable us to validate and tweak our technique for large-scale projects.

- Developing requirements consistency metrics
  Using the developed tools, one can design metrics to measure the consistency among requirements artifacts such as textual description and a set of use-cases. There are very few such metrics available for requirements engineering process and those too are more focused on traceability. Factors such as percentage of inconsistent classes or relations in the given requirements artifacts will provide a guideline for the metric development.
- Extending use-case grammar
  Since the use-case editor uses a context-free grammar to parse use-cases, the expressiblity is limited. We would benefit from a more powerful parsing technique which would allow us to provide more useful constructs.

Status of the project

- SVN repository: https://svn.cc.gatech.edu/jsr
- Website: http://www.cc.gatech.edu/projects/dowser
- Download: http://www.cc.gatech.edu/projects/dowser/download
- Requirements:
  - Ubuntu 9.04
  - Java 1.6
- Dependencies
  - Provided in the archive
    - UMLGraph 5.2
    - WordNet 3.0
    - ANTLR 3.0
    - 'jcalendar' API
    - JAWS API for accessing WordNet
  - Installed externally
    - Sun-Java6-JDK
    - 'plotutils' package
    - 'imagemagick' package
    - 'graphviz' package

References

1. *Transformation of SBVR Business Design to UML Models.* **Raj A., T.V. Prabhakar, Hendryx S.** 2008. pp. 29-38.

2. *GATE, General Architecture for Text Engineering.* **Cunningham, H.** 2, 2002, Vol. 36, pp. 223-254.

3. *(ACE), Attempto Controlled English.* **Fuchs N. E., Schwitter R.** Belgium : s.n., 1996. First International Workshop on Controlled Language Applications.

4. *Formalizing a Derivation Strategy for Formal Specifications from Natural Language Requirements Models.* **Mauco M.V., Leonardi M.C., Riesco D., Montejano G., Debnath N.** 2005. IEEE International Symposium on Signal Processing and Information Technology.

5. *Validating Use-Cases with the AsmL Test Tool.* **Barnett M., Grieskamp W., Schulte W., Tillmann N., Veanes M.** 2003. QSIC, IEEE Computer Society.

6. *Automated Conversion from Requirements Documentation to an Object-Oriented Formal Specification Language.*

Lee B. S., Bryant B. R. 2002. Symposium on Applied Computing. pp. 932 - 936.

7. *An approach for the synthesis of State transition graphs from Use Cases.* **S., Some S.** 2003. Proceedings of the International Conference on Software Engineering Research and Practice. Vol. I, pp. 456-462.

8. *Synthesizing Hierarchical State Machines.* **WHITTLE J., JAYARAMAN P. K.** 3, 2010, ACM Transactions on Software Engineering and Methodology, Vol. 19.

9. *Cross Checking Rules to Improve Consistency between.* **Ha I., Kang B.** 2008. Proceedings of the 9th International Conference on Intelligent Data Engineering and Automated Learning. Vol. 5326, pp. 436 - 443.

10. *On OO Design Consistency in Iterative Development.* **V., Bellur U. and Vallieswaran.** 2006. Proceedings of the Third International Conference on Information Technology: New Generations (ITNG). pp. 46-51.

11. *Information Extraction, Automatic.* **H., Cunnigham.** 2006, Encyclopedia of Language & Linguistics, Second Edition, Vol. 5, pp. 665-677.

12. *Object-Oriented Analysis: Getting help from robust computational linguistic tools.* **Delisle S., Barker K., Biskri I.** 1999. Application of Natural Language to Information Systems. pp. 167–172.

13. *Ambiguity in Requirements Specification, Perspectives on Requirements Engineering.* **Berry D.M., Kamsties E.** 2004. pp. 7-44.

14. **Fuchs N.E., Schwertel U., Schwitter R.** *Attempto controlled english (ACE) language manual, version 3.0.* 1999.

15. **B., Gervasi V. and Nuseibeh.** Lightweight validation of natural language requirements. *Software—Practice & Experience.* 2002, Vol. 32, 2, pp. 113-133.

16. *An environment for use-cases based requirements engineering.* **Some, Stephane.** 2004. 12th IEEE International Requirements Engineering Confer-. pp. 364-365.

17. *Improving the quality of natural language requirements specifications through natural language requirements patterns.* **Tjong S.F., Hallam N., and Hartley M.** 2006. 6th IEEE International confer-. pp. 199-199.

18. *An approach for matching functional business requirements to standard application software packages via ontology.* **Kluge R., Hering T., Belter R., and Franczyk B.** 2008. Computer Software and Applications, 32nd Annual IEEE International conference. pp. 1017-1022.