LinTest, A development tool for testing dialogue systems

Lars Degerstedt and Arne Jönsson

LINTest, A development tool for testing dialogue systems

Lars Degerstedt, Arne Jönsson

Department of Computer and Information Science
Linköpings universitet, Linköping, Sweden
`larde@ida.liu.se, arnjo@ida.liu.se`

Abstract

In this paper we present a development tool for testing dialogue systems. Testing software through the specification is important for software development in general and should be as automated as possible. For dialogue systems, the corpus can be seen as one part of the specification and the dialogue system should be tested on available corpora on each new build. The testing tool is inspired from work on agile software development methods, test driven development and unit testing, and can be used in two modes and during various phases of development.
Index Terms: dialogue systems, development tools, corpus, testing.

1. Introduction

Natural Language Processing (NLP) and Language Engineering (LE) are two complementary branches of the field of language technology aiming at similar goals. Following Cunningham [1] "NLP is part of the science of computation whose subject matter is computer systems that process human language"[1, p. 4] whereas "Language Engineering is the discipline or act of engineering software systems that perform tasks involving processing human language" [1, p. 5]. Thus, although both NLP and LE are studying processing of human language, there is a difference in focus. For LE the primary goal is on developing software products that are stable, robust and effective whereas NLP focuses on investigations of computational means of more theoretical nature, for processing human language.

From a language engineering perspective the goal is not only to construct language technology but to develop it following the principles of engineering—measurable and predictable systems, methodologies, and resource bounds [1]. Factors of reuse, effective development, and guaranteed quality then becomes important issues. The primary goal of language engineering is developing software products that are stable, robust and effective. For the language engineer there is thus a need to re-use previously developed language components and to make them gradually more generic. This is important not only for industry projects but also for research projects.

The work presented in this paper is based on iterative development [2, 3, 4] and an evolutionary view on project development, cf. [5], much resembling the ideas advocated by the agile manifesto[1].

The agile methodology facilitates robust handling of surprises and changes during the course of a project. It emphasises the need for collaboration between team players with different background and purpose [6]. Moreover, it is highly influential on the new software development tools of today and the future, such as strong emphasis on unit testing and code refactoring. The common mainstream software theory is thereby also becoming more agile and so the agile technology will by necessity influence a related field such as language technology.

When building an advanced interactive system, we are working with sparse knowledge models in new territory due to the inherent complexity in the underlying communicative processes. For dialogue systems, we typically use a new mixture of different theories in new blends for each new system, in order to deal with the complexity and uniqueness of natural language. We believe that the agile methodology has much to offer such unforeseeable development processes, and also that natural language processing is an interesting border-case to study for the agile development community.

The agile pieces and project competences must all fit together for the team to work well on all aspects. Both the individual and the team should strive to become a Jack of All Trades [7] to be able to handle the inherently unforeseeable situation during development. This includes handling surprises on both linguistic and software level well, in an unforeseeable and rapidly changing system construction process. The team must embrace change on all levels, just as the natural language itself does - it evolves.

Language engineering must consider both *adaptiveness* of the agile methodology for the software development (cf.[8]) and *cumulative language modelling* on the linguistic level.

Agile development emphasise unit testing and for many agile methods test driven development is important, e.g. to write the tests before the code. JUnit[2] is a tool developed to facilitate unit testing that is easy to use, that can be integrated in the normal build phase and that automatise testing of software components. With JUnit, users write test cases that are executed on each new build, to make sure that a certain component behaves as before.

2. Dialogue systems development

When developing dialogue systems it is crucial to have knowledge on how users will interact with the system. Such knowledge is often elicited from corpora, collected in various ways, natural, artificial or a mixture. The corpus is then used in the development of the dialogue system. From a testing perspective we identify three development activities:

- Specification, when new behaviour is implemented
- Refactoring, when the code is re-written without changing the external behaviour

[1]`http://agilemanifesto.org/`

[2]`http://www.junit.org/`

- Refinement, when the external behaviour is refined

The activities are not performed in any specific order and have mainly the purpose of illustrating our testing tool.

2.1. Specification

Specification includes all activities where new functionality is developed for the dialogue system. From a corpus perspective, initial development can be based on a subset of the corpus handling basic utterances, for instance, simple information requests. Already at this stage we see a need for a tool allowing for easy testing of the new features.

We also see a need for administration of the corpus dividing it into those utterances that can be handled by the current system, those that should have been handled but for some reason did not work and those that have not yet been considered. When more advanced features are to be incorporated into the dialogue system, this means that new corpora, or new subsets of existing corpora, are to be included in the test set.

2.2. Refactoring

Refactoring is the technique of changing the internal behaviour of the dialogue system without changing the external behaviour [9]. Refactoring is a common activity, especially in language engineering, where robust and generic code is in focus. Testing plays an important role in refactoring to ensure that everything that worked before works also after refactoring. Testing the re-factored system on all dialogues in the corpus that the system handled correctly before is therefore crucial. Such testing is preferably done automatically and in the background at each new build.

2.3. Refinement

With refinement we understand the process of fine-tuning the external behaviour of the system. We separate refinement from specification as the requirements on a testing tool differ. Specification means adding new behaviour and often include extending the corpus with dialogues covering the new phenomenon to be developed. Refinement also means changing the external behaviour, but only slightly. The new behaviour of the dialogue system is to be so close to the previous that the corpus in principle should work. Changing the prompt is a typical example of a refinement, i.e. the old corpus will no longer pass the test as the prompt is changed. Once the prompt in the corpus is changed to the new prompt the test will pass again. Thus, a minor update of the corpus is needed. The testing tool should allow for such update of the corpora reflecting the new refined behaviour.

3. LINTest

LINTest is a tool for testing dialogue systems against a corpus[3]. LINTest is inspired by the simplicity of JUnit, but it is *not* aimed at unit testing. It facilitates test-driven development of dialogue systems, but is intended for testing dialogue behaviour. LINTest is a tool to be easy to use for any dialogue systems developer and is less ambitious than interactive development environments such as GATE[4].

[3]LINTest is available for download from `http://nlpfarm.sourceforge.net/`

[4]`http://gate.ac.uk/`

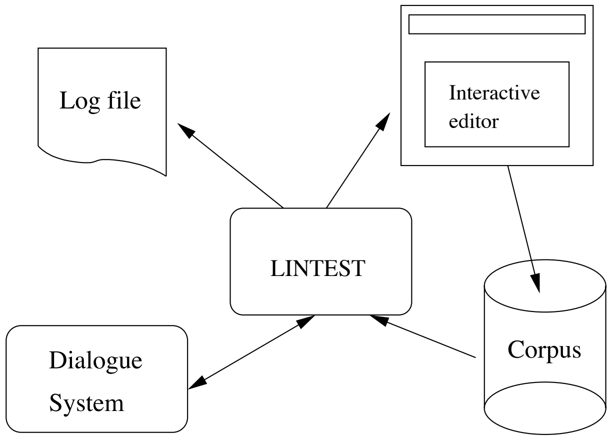


Figure 1: LINTest usage

LINTest runs a corpus collection on a dialogue system and compares the output from the dialogue system with the corpus. If the corpus thus produced is the same as the original corpus the dialogue system is assumed to behave according to current specification. LINTest only runs on typed interactions, but any transcribed spoken dialogue can in theory be used as long as the transcribed utterances' can be accepted by the dialogue system.

LINTest can be used in two different modes: batch mode and interactive mode, see figure 1. In batch mode nothing really happens as long as a test passes, the result from the test is presented in a log file. In interactive mode LINTest allows for more verbose testing and behaves more like a debugging tool.

Although both modes can be used in every phase of dialogue systems development, we believe that batch mode is more useful during refactoring and the interactive mode is more useful during refinement.

Testing during refactoring resembles unit testing in the sense that we want to ensure that previous external behaviour is preserved no matter how we rewrite the code. Testing should then interfere as little as possible, when all is fine.

Refinement, on the other hand, is about changing the external behaviour, and consequently, very few tests will pass. In interactive mode the software developer is presented with the result from running the modified system together with the corpus and can inspect effects of the refinement and also update the corpus to avoid having LINTest fail on further occasions of running the refined dialogue system.

Testing during specification adhere to the principle of writing the test code first [2]. Again, this is by no means unit testing, LINTest is used for system testing, i.e. the functionality of the whole system. In this phase of the development a mixture of interactive and batch mode is used. Initially, when new dialogues are added, interactive mode helps updating the corpus whereas batch mode allows for a less intrusive testing behaviour when developing a certain behaviour.

4. Illustration

To illustrate the use of LINTest we will briefly present how we have used LINTest in the development of BirdQuest [10].

```
run-lintest:
  [lintest] Dialog failed: 1141046899757(turn 4) utterance was Välj någon av följande lärkor:
            trädlärka, tofslärka, sånglärka och berglärka
  [lintest]  expected  Välj en av följande lärkor: trädlärka, tofslärka, sånglärka och berglärka

  [lintest] Dialog failed: 1141128937123(turn 2) utterance was  Välj någon av följande svanar:
            sångsvan, mindre sångsvan och knölsvan
  [lintest]  expected  Välj en av följande svanar: sångsvan, mindre sångsvan och knölsvan

  [lintest] Tests run: 20 Failures: 2 Errors: 0
```

Figure 3: Typical output from LINTest. (The numbers after `Dialog failed:` are automatically generated log-file names.)

```
<target name="run-lintest">
  <lintest agentclass="birdquest.TestAgent"
           showoutput="true"
           agentpath="${build.path}">
    <fileset dir="${source.path}/test/corpus"/>
  </lintest>
</target>
```

Figure 2: A typical target for testing BIRDQUEST using the dialogue system class `birdquest.TestAgent` with the corpus specified in the `fileset` parameter, presenting output on the screen. LINTest allows more parameters, e.g. to specify a file for output and to halt on failure.

The BIRDQUEST system is a dialogue system aimed at full support for basic dialogue capabilities, such as clarification subdialogues and focus management, developed for Swedish National Television (SVT), with information about Nordic birds. BIRDQUEST was developed from a corpus of 264 questions about birds collected by SVT on a web site for one of their nature programs, where the public could send in questions. We did unfortunately not have LINTest for this initial development.

BIRDQUEST is available on-line[5] and we have logged interactions for a number of years. The corpus currently (April 1 2006) comprise 284 dialogues with a total of 2046 user utterances (ranging from 1 to 56 user turns) (all interactions are in Swedish). This corpus has been used for refinement and refactoring of BIRDQUEST as briefly presented below. The presentation is aimed at demonstrating the use of LINTest.

4.1. Refactoring

As stated above, with refactoring we mean coding activities that do not change the external behaviour of the system. This ranges from minor changes, such as changing variable names, to re-implementing the whole system, for instance, in a new programming language.

Using LINTest during refactoring means that the systems external behaviour should be as before and we need means to run the test automatically on every build. Thus, LINTest is preferably used with a build tool. Currently we use ant and in the ant file specify the LINTest search path and additional parameters, in the target specification. An example entry is seen in Figure 2.

When building a new dialogue system with the target as specified in Figure 2 we get the usual build information and information on the number of test runs, normally that there were no errors or failures. A more interesting case is seen in Figure 3, where there are two errors, due to refinement.

4.2. Refinement

With refinement we understand changes to the system such that the corpus needs to be updated. Again consider Figure 3 which illustrates LINTest output when we change *Choose some of the following birds:* (*Välj någon av följande fåglar:* Swedish) to *Choose one of the following birds:* (*Välj en av följande fågar:* Swedish). This refinement of system behaviour means that the corpus will fail on many dialogues, and we therefore need to update the corpus to reflect this new system behaviour.

This done using the interactive editor in the upper right of Figure 1. Figure 4 depicts a screen shot of the interactive editor.

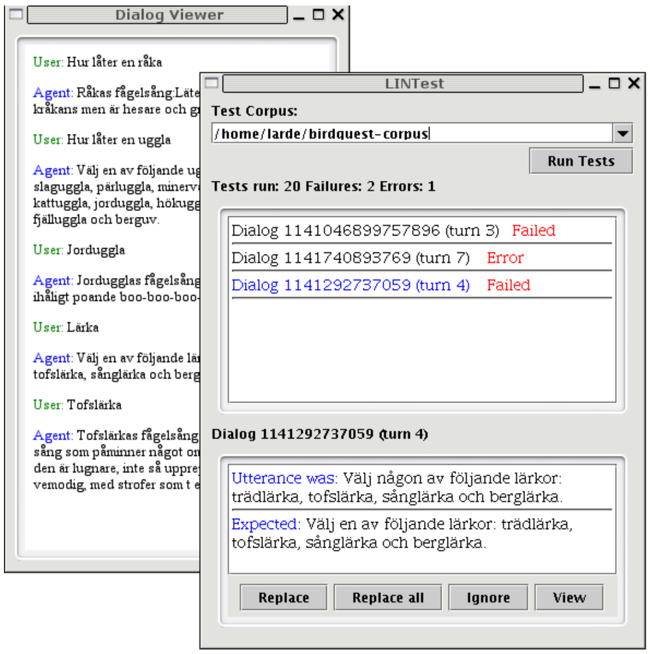
The interactive editor presents a result of the build and also displays utterances that differ. The user has four buttons. `Replace` means replacing the current utterance with the new. `Replace all` means replacing all occurrences of the error. By choosing `Replace all` the whole corpus is updated and when the developer continues testing, this particular refinement will not cause a failure. This option also allows for specifying which part(s) of the utterance to replace. The `Ignore` button in LINTest is used to mark errors as temporarily accepted.

4.3. Specification

When we analysed the corpus we found certain behaviours that BIRDQUEST could not handle [11]. Some of these errors are easy to handle, e.g. adding new words. Others involve more coding and also make more behaviour changes to the system. Both are examples of specification, meaning that we can not, as in refactoring expect that the system will have the same external behaviour nor, as in refinement, assume that we just want to change the corpus to reflect the new behaviour.

Instead, during specification we intertwine between batch and interactive testing. Often we start with a fairly small corpus subset and incrementally develop the new behaviour using LINTest in batch mode. Once the behaviour is near completion, or we for some other reason need to make sure that the system still handles previous dialogues in the corpus, we use LINTest on a larger por-

[5] `http://maj4.ida.liu.se:8080/birdquest`



User: Hur låter en råka

Agent: Råkas fågelsång:Läte
kråkans men är hesare och g

User: Hur låter en uggla

Agent: Välj en av följande ug
slaguggla, pärluggla, minerv
kattuggla, jorduggla, hökugg
fjälluggla och berguv.

User: Jorduggla

Agent: Jordugglas fågelsång
ihåligt poande boo-boo-boo-

User: Lärka

Agent: Välj en av följande lä
tofslärka, sånglärka och berg

User: Tofslärka

Agent: Tofslärkas fågelsång
sång som påminner något o
den är lugnare, inte så uppre
vemodig, med strofer som t e

Test Corpus: /home/larde/birdquest-corpus

Tests run: 20 Failures: 2 Errors: 1

Dialog 1141046899757896 (turn 3) Failed
Dialog 1141740893769 (turn 7) Error
Dialog 1141292737059 (turn 4) Failed

Dialog 1141292737059 (turn 4)

Utterance was: Välj någon av följande lärkor: trädlärka, tofslärka, sånglärka och berglärka.

Expected: Välj en av följande lärkor: trädlärka, tofslärka, sånglärka och berglärka.

Figure 4: Interactive editing in LINTest

tion of the corpus (or the whole corpus) in interactive mode to update the corpus with the new feature.

LINTest includes features to make a large corpus more tractable. It is possible to partition the corpus, using only those parts of the corpora that the are relevant for the current modification to the dialogue system. Partitions are especially useful when the corpus is large and we are adding new behaviour to the dialogue system as we then need a faster code-build-test loop. It is also possible to see a failure in context. Pressing the `View` button in the incremental editor brings up a dialogue viewer, see Figure 4.

5. Summary

In this paper we presented LINTest, a tool for testing dialogue systems from corpus data. LINTest origin from a need to support language engineering activities and is available for download as open source. LINTest facilitates incremental development of dialogue systems and can be used either in batch mode or in a GUI allowing for efficient corpus update.

6. Acknowledgements

This research was supported by Santa Anna IT Research Institute AB.

7. References

[1] Hamish Cunningham, "A Definition and Short History of Language Engineering," *Natural Language Engineering*, vol. 5, no. 1, pp. 1–16, 1999.

[2] Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 2000.

[3] Lars Degerstedt and Arne Jönsson, "A Method for Systematic Implementation of Dialogue Management," in *Workshop notes from the 2nd IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems, Seattle, WA*, 2001.

[4] Joris Hulstijn, *Dialogue Models for Inquiry and Transaction*, Ph.D. thesis, Universiteit Twente, 2000.

[5] M. M. Lehman and J. F. Ramil, "An Approach to a Theory of Software Evolution," in *Proc. of the 4th int. workshop on Principles of software evolution*, Vienna, Austria, September 2001.

[6] Alistair Cockburn and Jim Highsmith, "Agile software development: The people factor," *IEEE Computer*, pp. 131–133, November 2001.

[7] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.

[8] Jim Highsmith, *Adaptive Software Development*, Addison-Wesley, 1998.

[9] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Object Technology Series, 2000.

[10] Arne Jönsson, Frida Andén, Lars Degersedt, Annika Flycht-Eriksson, Magnus Merkel, and Sara Norberg, "Experiences from Combining Dialogue System Development with Information Access Techniques," in *New Directions in Question Answering*, Mark T. Maybury, Ed. AAAI/MIT Press., 2004.

[11] Annika Flycht-Eriksson and Arne Jönsson, "Some empirical findings on dialogue management and domain ontologies in dialogue systems – implications from an evaluation of birdquest," in *Proceedings of 4th SIGdial Workshop on Discourse and Dialogue, Sapporo, Japan*, 2003.