Context-Dependent Regression Testing for Natural Language Processing

Elaine Farrow
Human Communication Research Centre
School of Informatics
University of Edinburgh
Edinburgh, UK
Elaine.Farrow@ed.ac.uk

Myroslava O. Dzikovska
Human Communication Research Centre
School of Informatics
University of Edinburgh
Edinburgh, UK
M.Dzikovska@ed.ac.uk

Abstract

Regression testing of natural language systems is problematic for two main reasons: component input and output is complex, and system behaviour is context-dependent. We have developed a generic approach which solves both of these issues. We describe our regression tool, CONTEST, which supports context-dependent testing of dialogue system components, and discuss the regression test sets we developed, designed to effectively isolate components from changes and problems earlier in the pipeline. We believe that the same approach can be used in regression testing for other dialogue systems, as well as in testing any complex NLP system containing multiple components.

1 Introduction

Natural language processing systems, and dialogue systems in particular, often consist of large sets of components operating as a pipeline, including parsing, semantic interpretation, dialogue management, planning, and generation. Testing such a system can be a difficult task for several reasons. First, the component output may be context-dependent. This is particularly true for a dialogue system – reference resolution, ellipsis, and sometimes generation typically have to query the system state to produce their output, which depends both on the state of the world (propositions defined in a knowledge base) and on the dialogue history (object salience). Under these conditions, unit testing using the input and output of a single component in isolation is of limited value – the entire system state needs to be preserved to check that context-dependent components are functioning as expected.

Second, the inputs and outputs of most system components are usually very complex and often change over time as the system develops. When two complex representations are compared it may be difficult to determine what impact any change is likely to have on system performance (far-reaching or relatively trivial). Further, if we test components in isolation by saving their inputs, and these inputs are reasonably complex, then it will become difficult to maintain the test sets for the components further along the pipeline (such as diagnosis and generation) as the output of the earlier components changes during development.

The simplest way to deal with both of these issues would be to save a set of test dialogues as a gold standard, checking that the final system output is correct given the system input. However, this presents another problem. If a single component (generation, for example) malfunctions, it becomes impossible to verify that a component earlier in the pipeline (for example, reference resolution) is working properly. In principle we could also save the messages passing between components and compare their content, but then we are faced again with the problems arising from the complexity of component input and output which we described above.

To solve these problems, we developed a regression tool called CONTEST (for CONtext-dependent TESTing). CONTEST allows the authors of individual system components to control what information to record for regression testing. Test dialogues are saved and replayed through the system, and individual components are tested by comparing only their specific regression output, ignoring the outputs generated by other components. The components are isolated by maintaining a minimal set of inputs that are guaranteed to be processed correctly.

To deal with issues of output complexity we extend the approach of de Paiva and King (2008) for testing a deep parser. They created test sets at different levels of granularity, some including detailed representations, but some just saving very simple output of a textual entailment component. They showed that, given a carefully selected test set, testing on the final system output can be a fast and effective way to discover problems in the interpretation pipeline.

We show how the same idea can be used to test other dialogue system components as well. We describe the design of three different test sets that effectively isolate the interpretation, tutorial planning and generation components of our system. Using CONTEST allows us to detect system errors and maintain consistent test sets even as the underlying representations change, and gives us much greater confidence that the results of our testing are relevant to the performance of the system with real users.

The rest of this paper is organised as follows. In Section 2 we describe our system and its components in more detail. The design of the CONTEST tool and the test sets are described in Sections 3 and 4. Finally, in Section 5 we discuss how the interactive nature of the dialogue influences the design of the test sets and the process of verifying the answers; and we discuss features that we would like to implement in the future.

2 Background

This work has been carried out to support the development of BEETLE (Callaway et al., 2007), a tutorial dialogue system for basic electricity and electronics. The goal of the BEETLE system is to teach conceptual knowledge using natural language dialogue. Students interact with the system through a graphical user interface (GUI) which includes a chat interface,[1] a window to browse through slides containing reading material and diagrams, and an interface to a circuit simulator where students can build and manipulate circuits.

The system consists of twelve components altogether, including a knowledge base representing the state of the world, a curriculum planner responsible for the lesson structure, and dialogue management and NLP components. We developed CONTEST so that it could be used to test any system component, though our testing focuses on the natural language understanding and generation components.[2]

BEETLE uses a standard natural language processing pipeline, starting with a parser, lexical interpreter, and dialogue manager. The dialogue manager handles all input from the GUI (text, button presses and circuits) and also supports generic dialogue processing, such as dealing with interpretation failures and moving the lesson along. Student answers are processed by the diagnosis and tutorial planning components (discussed below), which function similarly to planning and execution components in task oriented dialogue systems. Finally, a generation subsystem converts the semantic representations output by the tutorial planner into the final text to be presented to the student.

The components communicate with each other using the Open Agent Architecture (Martin et al., 1998). CONTEST is implemented as an OAA agent, accepting requests to record messages. However, OAA is not essential for the system design – any communication architecture which supports adding extra agents into a system would work equally well.

BEETLE aims to get students to support their reasoning using natural language, since explanations and contentful talk are associated with learning gain (Purandare and Litman, 2008). This requires detailed analyses of student answers in terms of correct, incorrect and missing parts (Dzikovska et al., 2008; Nielsen et al., 2008). Thus, we use the TRIPS parser (Allen et al., 2007), a deep parser which produces detailed analyses of student input. The lexical interpreter extracts a list of objects and relationships mentioned, which are checked against the expected answer. These lists are fairly long – many expected answers have ten or more relations in them. The

[1] The student input is currently typed to avoid issues with automated speech recognition of complex utterances.

[2] All our components are rule-based, but we expect the same approach would work for components of a statistical nature.

diagnoser categorises each of the objects and relationships as correct, contradictory or irrelevant. The tutorial planner makes decisions about the remediation strategy, choosing one strategy from a set of about thirteen depending on the question type and tutorial context. Finally, the generation system uses the FUF/SURGE (Elhadad, 1991) deep generator to generate feedback automatically.

Obviously, the output from the deep parser and the input to the tutorial planner and generator are quite complex, giving rise to the types of problems that we discussed in the introduction. We already had a tool for unit-testing the parser output (Swift et al., 2004), plus some separate tools to test the diagnoser and the generation component, but the complexity of the representations made it impractical to maintain large test sets that depended on such complex inputs and outputs. We also wanted a unified way to test all the components in the context of the entire system. This led to the creation of CONTEST, which we describe in the rest of the paper.

3 The CONTEST Tool

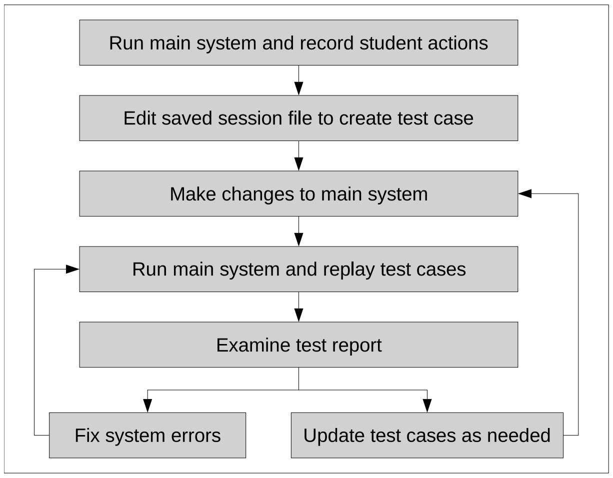
Run main system and record student actions
Edit saved session file to create test case
Make changes to main system
Run main system and replay test cases
Examine test report
Fix system errors
Update test cases as needed

Figure 1: The regression testing process.

In this section we describe the process for creating and using test cases, illustrated in Figure 1. The first step in building a useful regression tool is to make it possible to run the same dialogue through the system many times without retyping the student answers. We added a wrapper around the GUI to intercept and record the user actions and system calls for later playback, thus creating a complete record of the session. Every time our system runs, a new saved session file is automatically created and saved in a standard location. This file forms the basis for our test cases. It uses an XML format, which is human-readable and hand-editable, easily extensible and amenable to automatic processing. A (slightly simplified) example of a saved session file is shown in Figure 2. Here we can see that a slide was displayed, the tutor asked the question "Which components (if any) are in a closed path in circuit 1?" and the student answered "the battery and the lightbulb".

Creating a new test case is then a simple matter of starting the system and performing the desired actions, such as entering text and building circuits in the circuit simulator. If the system is behaving as it should, the saved session file can be used directly as a test case. If the system output is not as desired, the file can be edited in any text editor.

Of course, this only allows the final output of the system to be tested, and we have already discussed the shortcomings of such an approach: if a component late in the pipeline has problems, there is no way to tell if earlier components behaved as expected. To remedy this, we added a mechanism for components other than the GUI to record their own information in the saved session file.

Components can be tested in effective isolation by combining two mechanisms: carefully designed test sets which focus on a single component and (importantly) are expected to succeed even if some other component is having problems; and a regression tool which allows us to test the output of an individual component. Our test sets are discussed in detail in Section 4. The remainder of this section describes the design of the tool.

CONTEST reads in a saved session file and reproduces the user actions (such as typing answers or building circuits), producing a new saved session file as its output. If there have been changes to the system since the test was created, replaying the same actions may lead to new slides and tutor messages being displayed, and different recorded output from intermediate components. For example, given the same student answers, the diagnosis may have changed, leading to different tutor feedback. We compare the newly generated output file against the input file. If there are no differences, the test is considered to have passed. As the input and output files

```
<test>
  <action agent="tutor" method="showSlide">
    lesson1-oe/exercise/img1.html
  </action>
  <action agent="tutor" method="showOutput">
    Which components (if any) are in a closed path in circuit 1?
  </action>
  <action agent="student" method="submitText">
    the battery and the lightbulb
  </action>
</test>
```

Figure 2: A saved session file showing a single interaction between tutor and student.

are identical in format, the comparison can be done using a 'diff' command.

With each component recording its own output, it can be the case that there are many differences between old and new files. It is therefore important to be able to choose the level of detail we want when comparing saved session files, so that the output of a single component can be checked independently of other system behaviour. We solved this problem by creating a set of standard XSLT filters. One filter picks out just the dialogue between student and tutor to produce a transcript of the session. Other filters select the output from one particular component, for example the tutorial planner, with the tutor questions included to provide context. In general, we wrote one filter for each component.

CONTEST creates a test report by comparing the expected and actual outputs of the system on each test run. We specify which filter to use (based on which component we are testing). If the test fails, we can examine the relevant differences using the 'ediff' mode in the emacs text editor. More sophisticated approaches are possible, such as using a further XSL transform to count all the errors of a particular type, but we have found ediff to be good enough for our purposes. With the filters in place we only see the differences for the component we are testing. Since component regression output is designed to be small and human-readable, checking the differences is a very quick process.

Test cases can be run individually or in groups.[3] Using CONTEST, we can create a single report for a group of test cases: the individual outputs are combined to create a new output file for the group and this is compared to the (combined) input file, with filters applied in the usual way. This is a very useful feature, allowing us to create a report for all the 'good answer' cases (for example) in one step.

Differences do not always indicate errors; sometimes they are simply changes or additions to the recorded information. After satisfying ourselves that the reported differences are intentional changes, we can update the test cases to reflect the output of the latest run. Subsequent runs will test against the new behaviour. CONTEST includes an update tool which can update a group of cases with a single command. This is simpler and less error-prone than editing potentially hundreds of files by hand.

4 Test Cases

We have built several test sets for each component, amounting to more than 400 individual test cases. We describe examples of the test sets for three of our components in more detail below, to demonstrate how we use CONTEST.

4.1 Interpretation Test Cases

We have a test set consisting of 'good answers' for each of the questions in our system which we use to test the interpretation component. The regression information recorded by the interpretation component includes the internal ID code of the matched answer and a code indicating whether it is a 'best', 'good' or 'minimal' answer. This is enough to allow us to de-

[3]Test cases are usually grouped by directory, but symbolic links allow us to use the same case in several groups.

```
<test name="closed_path_discussion">
  <action agent="tutor" method="showOutput">
    What are the conditions that are required to make a bulb light up?
  </action>
  <action agent="student" method="submitText">
    a bulb must be in a closed path with the battery
  </action>
  <action agent="simpleDiagnosis" method="logForRegression">
    student-act: answer atype: diagnosis consistency: []
    code: complete subcode: best
    answer_id: conditions_for_bulb_to_light_ans1
  </action>
</test>
```

Figure 3: A sample test case from our 'good answers' set showing the diagnosis produced for the student's answer.

tect many possible errors in interpretation. We can run this test set after every change to the parsing or interpretation components.

A (slightly simplified) example of our XML test case format is shown in Figure 3, with the tutor question "What are the conditions that are required to make a bulb light up?" and the student answer "a bulb must be in a closed path with the battery". The answer diagnosis shows that the system recognised that the student was attempting to answer the question (rather than asking for help), that the answer match was complete, with no missing or incorrect parts, and the answer was consistent with the state of the world as perceived by the system.[4] The matched answer is marked as the best one for that question.

While the recorded information does not supply the full interpretation, it can suggest the source of various possible errors. If interpretation fails, the student act will be set to `uninterpretable`, and the `code` will correspond to the reason for failed interpretation: `unknown_input` if the parse failed, `unknown_mapping` or `restriction_failure` if lexical interpretation failed, and `unresolvable` if reference resolution failed. If interpretation worked, but took incorrect scoping or attachment decisions, the resulting proposition is likely to be inconsistent with the current knowledge base, and an inconsistency code will be reported. In addition, verifying the matched answer ID provides some information in case only a partial interpretation was produced. Sometimes different answer IDs correspond to answers that are very complete versus answers that are acceptable because they address the key point of the question, but miss some small details. Thus if a different answer ID has matched, it indicates that some information was probably lost in interpretation.

The codes we report were not devised specifically for the regression tests. They are used internally to allow the system to produce accurate feedback about misunderstandings. However, because they indicate where the error is likely to originate (parsing, lexical interpretation, scoping and disambiguation), they can help us to track it down.

We have another test set for 'special cases', such as the student requesting a hint or giving up. An example is shown in Figure 4. Here the student gives up completely on the first question, then asks for help with the second. We use this test case to check that the set phrases "I give up" and "help" are understood by the system. The 'special cases' test set includes a variety of help request phrasings observed in the corpora we collected. Note that this example was recorded while using a tutorial policy that responds to help requests by simply providing the answer. This does not matter for testing interpretation, since the information recorded in the test case will distinguish help requests from give ups, regardless

[4]Sometimes students are unable to interpret diagrams, or are lacking essential background knowledge, and therefore say things that contradict the information in the domain model. The system detects and remediates such cases differently from general errors in explanations (Dzikovska et al., 2006).

T: Which components (if any) are in a closed path in circuit 1?
S: I give up
T: The answer is the battery and the bulb in 1.
T: Which components (if any) are in a closed path in circuit 2?
S: help
T: Here's the answer. The bulb in 2.

Figure 4: The transcript of a test case for "I give up" and "help". **T:** is the tutor, **S:** is the student.

of the specific tutorial policy used by the system.

Finally, we have a test set for detection of uninterpretable utterances. In a tutoring system, students often use incorrect terminology or unexpected phrasings which the system cannot understand. While we expect coverage and robustness of parsing and interpretation components to improve as the system develops, some percentage of errors is to be expected simply because even state-of-the-art parsers are not 100% accurate. Thus the system needs to detect problems whenever possible, and produce useful help messages. We have collected a set of student utterances that correspond to typical interpretation problems – failed parses, unknown words, sentences that are parsed but the parser makes an incorrect attachment decision, sentences where the student used terminology incorrectly (violating sortal constraints encoded in the knowledge base), a pronominal referent that could not be resolved. Identifying the problem type correctly is important because it affects later processing in the dialogue manager and tutorial planner. Thus we have a set of test cases using the collected sentences, which we use to ensure that the problems are flagged correctly.

4.2 Tutoring Test Cases

The BEETLE tutorial planner makes decisions on system behaviour and corresponds to planning and execution components in other dialogue systems. Here, we are attempting to test that the tutorial planner selected the right system action. Therefore, since we are not testing the interpretation component, we don't vary the form of the language we use. Instead,

T: Predict the state (on, off) of each bulb in diagram 1.
S: all bulbs are off
T: OK, we'll check your prediction in a moment.
T: Predict the state (on, off) of each bulb in diagram 2.
S: all bulbs are off
T: OK, we'll check your prediction in a moment.
T: Test your prediction for diagram 1 by opening both switches and observing the bulbs.
T: Test your prediction for diagram 2 by closing both switches and observing the bulbs.
T: Did the bulbs behave the way you expected?
S: no
T: Right. In your second prediction, you said that Bulb A, Bulb B and Bulb C would be off. But Bulb A, Bulb B and Bulb C are lit.

Figure 5: A tutorial planner test case transcript. **T:** is the tutor, **S:** is the student.

we vary the combination of good and bad student answers, and record the action chosen by the system.

The tutorial planner chooses feedback in 2 stages. First, a general algorithm is chosen depending on the exercise type and student input type: there are separate algorithms for addressing, for example, what to do if the student input was not interpreted, and for correct and incorrect answers. Choosing the algorithm requires some computation depending on the question context. Once the main algorithm is chosen, different tutorial strategies can be selected, and this is reflected in the regression output: the system records a keyword corresponding to the chosen algorithm, and then the name of the strategy along with key strategy parameters.

For example, Figure 5 shows the transcript from a test case for a common exercise type from our lessons: a so called predict-verify-evaluate sequence. In this example, the student is asked to predict the behaviour of three light bulbs in a circuit, test it by manipulating the circuit in the simulation environment, and then evaluate whether their predictions matched the circuit behaviour. The system reinforces the point of the exercise by producing a summary of discrepancies between the student's

```
<action agent="tutor" method="showOutput">
  Did the bulbs behave the way you expected?
</action>
<action agent="student" method="submitText">
  no
</action>
<action agent="tc-bee" method="logForRegression">
  EVALUATE (INCORRECT-PREDICTION NO_NO)
</action>
```

Figure 6: An excerpt from a tutorial planner test case showing the recorded summary output.

predictions and the observed outcomes.

An excerpt from the corresponding test case is shown in Figure 6. Here we can see the tutor ask the evaluation question "Did the bulbs behave the way you expected?" and the student answer "no". The `EVALUATE` algorithm was chosen to handle the student answer, and from the set of available strategies the `INCORRECT-PREDICTION` strategy was chosen. That strategy takes a parameter indicating if there was a discrepancy when the student evaluated the results (here `NO_NO`, corresponding to the expected and actual evaluation result inputs).

In contrast, in the first example in Figure 4, where the student gives up and doesn't provide an answer, the tutorial planner output is `REMEDIATE (BOTTOM-OUT Q_IDENTIFY)`. This shows that the system has chosen to use a `REMEDIATE` algorithm, and a 'bottom-out' (giving away the answer) strategy for remediation. The strategy parameter `Q_IDENTIFY` (which depends on the question type) determines the phrasing to be used in the generator to verbalise the tutor's feedback.

The saved output allows us to see that the correct algorithm was chosen to handle the student input (for example, that the `REMEDIATE` algorithm is correctly chosen after an incorrect student answer to an explanation question), and that the algorithm chooses a strategy appropriate for the tutorial context. Certain errors can still go undetected here, for example, if the algorithm for verbalising the chosen strategy in the generator is broken. Developing summary inputs to detect such errors is part of planned future work.

In order to isolate the tutorial planner from interpretation, we use standard fixed phrasings for student answers. The answer phrasings in the 'good answers' test set for interpretation (described in Section 4.1) are guaranteed to be understood correctly, so we use only these phrasings in our tutorial planner test cases. Thus, we are able to construct tests which will not be affected by problems in the interpretation pipeline.

4.3 Generation Test Cases

To test generation, we have a set of test cases where the student immediately says "I give up" in response to each question. This phrase is used in our system to prevent the students getting stuck – the tutorial policy is to immediately stop and give the answer to the question. The answers given are generated by a deep generator from internal semantic representations, so this test set gives us the assurance that all relevant domain content is being generated properly. This is not a complete test for the generation capabilities of our system, since each explanation question can have several possible answers of varying degrees of quality (suggested by experienced human tutors (Dzikovska et al., 2008)), and we always choose the best possible answer when the student gives up. However, it gives us confidence that the student can give up at any point and receive an answer which can be used as a template for future answers.

5 Discussion and Future Work

We have created more than 400 individual test cases so far. There are more than 50 for the interpretation component, more than 150 for the tutorial planner and more than 200 for the generation component. We are developing new test sets based on other scenarios, such as responding to each question with a help request. We are also refining the summary information recorded by each component.

An important feature of our testing approach is the use of short summaries rather than the internal representations of component inputs and outputs. Well-designed summaries provide key information in an easy-to-read format that can remain constant as internal formats change and develop over time. We believe that this approach would be useful for other language processing systems, since at present there are few standardised formats in the community and representations are typically developed and refined together with the algorithms that use them.

The decision about what information to include in the summary is vital to the success and overall usefulness of the regression tool. If too much detail is recorded, there will be many spurious changes and it will be burdensome to keep a large regression set updated. If too little detail is recorded, unwanted changes in the system may go undetected. The content of the test cases we discussed in Section 4 represents our approach to such decisions.

Interpretation was perhaps the most difficult, because it has a particularly complex output. In determining the information to record, we were following the solution of de Paiva and King (2008) who use the decision result of the textual entailment system as a way to efficiently test parser output. For our system, the information output by the diagnoser about answer correctness proved to have a similar function – it effectively provides information about whether the output of the interpretation component was usable, without the need to check details carefully.

The main challenge for our tutorial planner and generation components (corresponding to planning and execution components in a task-oriented dialogue system) was to ensure that they were sufficiently isolated so as to be unaffected by errors in interpretation. We achieve this by maintaining a small set of known phrasings which are guaranteed to be interpreted correctly; this ensures that in practice, the downstream components are isolated from unwanted changes in interpretation.

Our overall methodology of recording and testing summary information for individual components can be used with any complex NLP system. The specific details of what information to record obviously depends on the domain, but our experience suggests some general principles. For testing the interpretation pipeline, it is useful to record pre-existing error codes and a qualitative summary of the information used to decide on the next system action. Where we record the code output by the diagnoser, an information seeking system could record, for example, the number of slots filled and the number of items retrieved from a database. It is also useful to record decisions taken by the system, or actions performed in response to user input; so, just as we record information about the chosen tutorial policy, other systems can record the action taken – whether it is to search the database, query a new slot, or confirm a slot value.

One major improvement that we have planned for the future is adding another layer of test case management to CONTEST, to enable us to produce summaries and statistics about the total number of test cases that have passed and failed, instead of checking reports individually. Such statistics can be implemented easily using another XSL transform on top of the existing filters to count the number of test cases with no differences and produce summary counts of each type of error detected.

6 Conclusion

The regression tool we developed, CONTEST, solves two of the major issues faced when testing dialogue systems: context-dependence of component behaviour and complexity of component output. We developed a generic approach based on running saved dialogues through the system, and checking summary information recorded by different components against separate gold standards. We demonstrated that test sets can be designed in such a way as to effectively isolate downstream components from changes and problems earlier in the pipeline. We believe that the same approach can be used in regression testing for other dialogue systems, as well as in testing any complex NLP system containing multiple components.

Acknowledgements

This work has been supported in part by Office of Naval Research grant N000140810043. We thank Charles Callaway for help with generation and tutoring tests.

References

James Allen, Myroslava Dzikovska, Mehdi Manshadi, and Mary Swift. 2007. Deep linguistic processing for spoken dialogue systems. In *Proceedings of the ACL-07 Workshop on Deep Linguistic Processing*.

Charles B. Callaway, Myroslava Dzikovska, Elaine Farrow, Manuel Marques-Pita, Colin Matheson, and Johanna D. Moore. 2007. The Beetle and BeeDiff tutoring systems. In *Proceedings of the SLaTE-2007 Workshop*, Farmington, Pennsylvania, USA, September.

Valeria de Paiva and Tracy Holloway King. 2008. Designing testsuites for grammar-based systems in applications. In *Coling 2008: Proceedings of the workshop on Grammar Engineering Across Frameworks*, pages 49–56, Manchester, England, August. Coling 2008 Organizing Committee.

Myroslava O. Dzikovska, Charles B. Callaway, and Elaine Farrow. 2006. Interpretation and generation in a knowledge-based tutorial system. In *Proceedings of EACL-06 workshop on knowledge and reasoning for language processing*, Trento, Italy, April.

Myroslava O. Dzikovska, Gwendolyn E. Campbell, Charles B. Callaway, Natalie B. Steinhauser, Elaine Farrow, Johanna D. Moore, Leslie A. Butler, and Colin Matheson. 2008. Diagnosing natural language answers to support adaptive tutoring. In *Proceedings 21st International FLAIRS Conference*, Coconut Grove, Florida, May.

Michael Elhadad. 1991. FUF: The universal unifier user manual version 5.0. Technical Report CUCS-038-91, Dept. of Computer Science, Columbia University.

D. Martin, A. Cheyer, and D. Moran. 1998. Building distributed software systems with the open agent architecture. In *Proceedings of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, Blackpool, Lancashire, UK.

Rodney D. Nielsen, Wayne Ward, and James H. Martin. 2008. Learning to assess low-level conceptual understanding. In *Proceedings 21st International FLAIRS Conference*, Coconut Grove, Florida, May.

Amruta Purandare and Diane Litman. 2008. Content-learning correlations in spoken tutoring dialogs at word, turn and discourse levels. In *Proceedings 21st International FLAIRS Conference*, Coconut Grove, Florida, May.

Mary D. Swift, Joel Tetreault, and Myroslava O. Dzikovska. 2004. Semi-automatic syntactic and semantic corpus annotation with a deep parser. In *Proceedings of LREC-2004*.