A Framework for the Generation of Computer System Diagnostics in Natural Language using Finite State Methods

Rachel Farrell
Dept Computer Science
University of Malta
rachelannefarrell@gmail.com

Gordon Pace
Dept Computer Science
University of Malta
gordon.pace@um.edu.mt

Michael Rosner
Dept Intelligent Computer Systems
University of Malta
mike.rosner@um.edu.mt

Abstract

Understanding what has led to a failure is crucial for addressing problems with computer systems. We present a meta-NLG system that can be configured to generate natural explanations from error trace data originating in an external computational system. Distinguishing features are the generic nature of the system, and the underlying finite-state technology. Results of a two-pronged evaluation dealing with naturalness and ease of use are described.

1 Introduction

As computer systems grow in size and complexity, so does the need for their verification. Whilst system diagnostics produced by automated program analysis techniques are understandable to developers, they may be largely opaque to less technical domain experts typically involved in scripting parts of the system, using domain-specific languages (Hudak, 1996) or controlled natural languages (CNLs) (Kuhn, 2014). Such individuals require higher level, less technical explanations of certain classes of program misbehaviour.

The problem boils down to an NLG challenge, starting from the trace (representing a history of the system) and yielding a narrative of the behaviour at an effective level of abstraction. The choice of an appropriate level of abstraction is particularly challenging since it is very dependent on the specification being matched or verified.

Pace and Rosner (Pace and Rosner, 2014), showed how a finite-state (FS) system can be used to generate effective natural language descriptions of behavioural traces. Starting from a particular property, they show how more natural and abstract explanations can be extracted from a system trace violating that property. However, the approach is manual and thus not very feasible for a quality assurance engineer. We show how their approach can be generalised to explain violations of general specifications. Since the explanation needs to be tailored for each particular property, we develop a general system, fitting as part of a verification flow as shown in Fig. 1. Typically, a quality assurance engineer is responsible for the top part of the diagram — giving a property specification which will be used by an analysis tool (testing, runtime verification, static analysis, etc) to try to identify violation traces. With our approach, another artefact

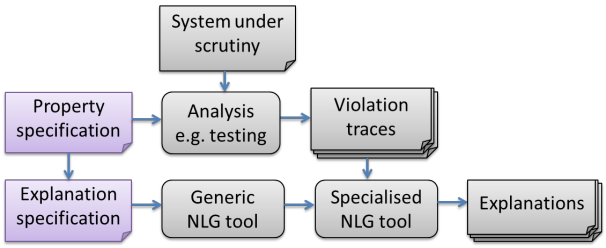

Figure 1: The architecture for general system diagnostics

is required, the *explanation specification*, which embodies the domain-specific natural language information for the property in question. From this, a generic NLG tool produces a specialised generation tool (embodying the domain-specific information and general information implicit in the traces) which can produce explanations for violations of that property. Our techniques have been implemented in a generic NLG tool, for which we show that the cost of adding user explanations for a property at an appropriate level of abstraction and naturalness is very low especially when compared to the cost of extending the system to identify such behaviours (e.g. developing test oracles or expressing a property using a formal language). The main novelty has been to develop a framework for generalising the approach developed earlier. We also further substantiate the claim that there is a place for FS methods in NLG.

2 Trace Explanation Styles

For explanations we adopted a CNL approach. The target language comprises (i) domain-specific terms and notions particular to the property being violated by the traces; and (ii) terms specific to the notions inherent to traces — such as the notions of events (as occurrences at points in time) and temporal sequentiality (the trace contains events ordered as they occurred over time). Following Pace and Rosner, we identify a sequence of progressively more sophisticated explanations of a particular violation trace. To illustrate this, consider an elevator system which upon receiving a request for the lift from a particular floor (<r1>–<r4>), services that floor by moving up or down (<u>, <d>). Once the lift arrives at a particular floor (<a1>–<a4>), the doors open (<o>). The

Proceedings of the 15th European Workshop on Natural Language Generation (ENLG), pages 52–56,
Brighton, September 2015.

doors can then either close (<c>) automatically, or after a floor request. Monitoring the property that the lift should not move with an open door, we will illustrate explanations with different degrees of sophistication of the violation trace: <a4,o,r4,a4,r2,c,d,a3,d,a2,o,r3,u>.

The simplest explanation is achieved in CNL0, where every symbol is transformed into a separate sentence, with an additional sentence at the end giving the reason why a violation occurred.

CNL0
The lift arrived at floor 4. The doors opened. A user requested to go to floor 4. The lift arrived at floor 4. A user requested to go to floor 2. The doors closed. The lift moved down. The lift arrived at floor 3. The lift moved down. The lift arrived at floor 2. The doors opened. A user requested to go to floor 3. The lift moved up. However this last action should not have been allowed because the lift cannot move with open doors.

In CNL1, the text is split into paragraphs consisting of sequences of sentence:

CNL1
1. The lift arrived at floor 4. 2. The doors opened. A user requested to go to floor 4. The lift arrived at floor 4. 3. A user requested to go to floor 2. The doors closed. The lift moved down. The lift arrived at floor 3. The lift moved down. The lift arrived at floor 2. 4. The doors opened. A user requested to go to floor 3. The lift moved up. However this last action should not have been allowed because the lift cannot move with open doors.

In CNL2, aggregation (Dalianis, 1999) techniques combine the single clause sentences from the previous two realisations to build multi-clause sentences, thus eliminating redundancy achieved through (i) the use of commas and words such as 'and', 'then', 'but' or 'yet', and (ii) the grouping of similar events, for example by stating the number of occurrences (e.g. 'moved down two floors').

CNL2
1. The lift arrived at floor 4. 2. The doors opened and a user requested to go to floor 4, yet the lift was already at floor 4. 3. A user requested to go to floor 2, then the doors closed. The lift moved down two floors and arrived at floor 2. 4. The doors opened, a user requested to go to floor 3, and the lift moved up. However this last action should not have been allowed because the lift cannot move with open doors.

Since the explanation contains detail which may be unnecessary or can be expressed more concisely, CNL3 uses summarisation — for instance, the first sentence in the explanation below summarises the contents of what were previously paragraphs 1–3. The last paragraph is left unchanged, since every sentence included there is required to properly understand the cause of the error.

CNL3
1. The lift arrived at floor 4, serviced floor 4, then serviced floor 2. 2. The doors opened, a user requested to go to floor 3, and the lift moved up. However this last action should not have been allowed because the lift cannot move with open doors.

For Pace and Rosner the explanation language is a CNL, whose basis, described in the Xerox Finite State Toolkit (XFST) (Beesley and Karttunen, 2003) by a human author, states how system trace actions should be expressed. The natural language explanation is obtained by composing FS transducers in a pipeline. FS technologies are best-known for the representation of certain kinds of linguistic knowledge, most notably morphology (Wintner, 2008). In contrast, we used XFST to implement linguistic techniques such as structuring the text into paragraphs, aggregation, contextuality — as previously illustrated.

3 Generalised Explanations

Given a particular property, one can design a NLG tool capable of explaining its violation traces. Some of the explanation improvements presented in the previous section are common to most properties. We thus chose to address the more general problem of trace violation explanations, such that, although domain-specific concepts (e.g. the meaning of individual events and ways of summarising them) need to be specified, much of the underlying machinery pertaining to the implied semantics of the event traces (e.g. the fact that a trace is a temporally ordered sequence of events, and that the events are independent of each other) will be derived automatically. The resulting approach, as shown in Fig. 1, in which we focus on the *Generic NLG* component uses the domain-specific information about a particular property (the *Explanation Specification* script provided by a QA engineer) to produce an explanation generator for a whole class of traces (all those violating that property). A specification language was created to facilitate the creation of a specification by non-specialist users. A script in the general trace-explanation language is used to automatically construct a specific explanation generator in XFST, going beyond a NLG system by developing a generator of trace explanation generators.

4 Specifying Trace Explanations

Scripts for our framework allow the user to specify the domain-specific parts of the explanations for a particular property, leaving other generic language features to be deduced automatically. The core features of the scripting language are:

Explaining events: Rather than give a complete sentence for each event represented by a symbol, we split the information into the *subject* and *predicate*, enabling us to derive automatically when sequential actions share a subject (thus allowing their combination in a more readable form). For example, the EXPLAIN section of the script is used to supply such event definitions:

```
EXPLAIN {
 <a4>: {
    subject:   "the lift";
    predicate: "arrived at level four";
 }
 ...
}
```

Events in context: Certain events may be better explained in a different way in a different context. For instance, the event `a4` would typically be described as 'The lift arrived at floor four', except for when the lift is already on the fourth floor, when one may say that 'The lift remained at floor four'. Regular expressions can be used to check the part of the trace that precedes or follows a particular event to check for context:

```
<a4>: {
   subject:   "the lift";
   predicate {
     context: {
       default: "arrived at level four";
       <r4>_ :  "remained at floor four";
     }
   }
}
```

Compound explanations: Sometimes, groups of symbols would be better explained together rather than separately. Using regular expressions, the `EXPLAIN` section of the script allows for such terms to be explained more succinctly:

```
<r2><c><d><a3><d><a2>: {
  subject:   "the lift";
  predicate: "serviced floor 2";
}
```

Errors and blame: Errors in a violation trace typically are the final event in the trace. We allow not only for the description of the symbol in this context, but also an explanation of what went wrong and, if relevant, where the responsibility lies:

```
ERROR_EXPLAIN {
  [<u>|<d>]: {
    blame: "due to a lift controller malfunction";
    error_reason:
      context: {
        default: "";
        [<o>[<r1>|<r2>|<r3>|<r4>]]_:
           "the lift cannot move with open doors";
      }
 }
}
```

Document structure: A way is needed to know how to structure the document by stating how sentences should be formed and structured into paragraphs. Using CNL1 as an example, we can add a newline after the lift arrives at a floor. Similarly, based on the example for CNL2, we specify that the event sequence `<o><r4><4>` should be aggregated into a (enumerated) paragraph:

```
SENTENCE_AGGREGATION{
  [<1>|<2>|<3>|<4>]: { newline: after; }
  <o><r4><4>;
}
```

5 Evaluation

Two aspects of our approach were evaluated: (i) How much effort is required to achieve an acceptable degree of naturalness, and (ii) How difficult it is for first time users to write specifications.

5.1 Effort In-Naturalness Out

Since, using our framework a degree of naturalness can be achieved depending on the complexity of the logic encoded in our script, unsatisfactory explanations may be caused by limitations of our approach or just a badly written script. The framework was first evaluated to assess how effort put into writing the script for a particular property correlates with naturalness of the explanations.

To measure this, we considered properties for an elevator controller, a file system and a coffee vending machine. We then built a series of scripts, starting with a basic one and progressively adding more complex features. For each property, we thus had a sequence of scripts of increasing complexity, where the time taken to develop each was known. These scripts were then executed in our framework on a number of traces, producing a corpus of natural language explanations each with the corresponding trace and associated script development time. The sentences together with the corresponding trace (but not the script or time taken to develop it) were then presented using an online questionnaire to human judges who were asked to assess the naturalness, readability and understandability of the generated explanations.

Explanations were rated on a scale from 1–6[1]. Evaluators were presented with a fraction of the generated explanations, shown in a random order, to prevent them from making note of certain patterns, which might have incurred a bias. Over 477 responses from around 64 different people.

The results of this analysis can be found in Table 1, which shows the scores given to explanations for the different systems and for traces produced by the scripts with different complexity. The results show that the naturalness of the generated explanations was proportional to the time taken to write the scripts — the best-faring explanations having a high rate of aggregation and summarisation. Interestingly, even with scripts written quickly e.g. 15–20 minutes[2] many evaluators still found the explanations satisfactory.

Figure 2 shows the results of plotting time taken to write the script (x-axis) against naturalness of the explanation (y-axis). For the coffee machine and elevator controller traces, the graphs begin to stabilise after a relatively short time, converging to a limit 80% of which is roughly achieved during the first 20–30% of the total time taken to create the most comprehensive script we wrote. The graph for the file system traces gives a somewhat different view; a higher overall score is obtained, yet we do not get the same initial gradient steepness[3]. A reason for the discrepancy in the graph shape could be that traces obtained for this system contained many repeated symbols in succession, hence until a script handled this repetition, the explanations received low scores. This shows that there may exist a relation between the kind of system being considered and the effort and linguistic techniques required to generate natural sounding explanations for its traces.

[1] From 1–6: unnatural and difficult to follow, unnatural but somewhat easy to follow, unnatural but very easy to follow, contains some natural elements, fairly natural, very natural and easy to follow.

[2] Recall that one script can be used to explain any counter-example trace for that property, and would thus be repeatedly and extensively used during system verification or analysis.

[3] It is worth noting that, for example, the first data point in all graphs occurs at the time after which similar linguistic techniques were included in the script.

Table 1: Overall scores given to generated explanations

System	Time /mins	Score								
		1	2	3	4	5	6	Mean	Mode	Median
Elevator system	10	1	8	10	9	2	10	3.83	3,6	4
	16	2	4	4	9	15	9	4.35	5	5
	24	1	2	2	4	15	6	4.6	5	5
	39	1	0	3	8	8	11	4.77	6	5
File system	12	5	7	11	3	3	1	2.83	3	3
	19	5	8	7	7	8	7	3.62	2,5	4
	22	2	5	13	5	6	3	3.5	3	3
	32	0	2	4	5	14	18	4.98	6	5
Coffee machine	10	3	4	4	12	5	8	4	4	4
	15	3	6	4	8	14	4	3.92	5	4
	25	1	3	5	3	9	8	4.38	5	5
	28	1	1	3	10	10	11	4.67	6	5
	38	2	1	2	4	18	17	4.95	5	5

We can thus conclude that whilst a certain inherent limit exists, natural-sounding explanations can be well achieved using this system. Effort however is rather important, and usually, the more time invested in building a script, the better the quality of the output. Nevertheless, even with minimal effort, a script author highly familiar with the input language can obtain a rather satisfactory output.

5.2 User Acceptance Test

To assess the framework's accessibility, we ran a four-hour experiment with four new users familiar with concepts such as regular expressions. They were requested to produce scripts to explain different properties unaided and were then asked to rate the ease of use and expressivity of the input language, their satisfaction with the output generated, and whether it matched their expectations. Given the low number of participants, the results are only indicative, and assessing the quality of the scripts they produced would not have given statistically meaningful results.

Overall, these users characterised the scripting language between *somewhat difficult* to *easy to use*. Dealing with contextual explanation of events presented the greatest challenges, although all managed to produce an error explanation which required using this concept. Apart from simply explaining every symbol as a complete sentence, the users also managed to create scripts involving aggregation and summarisation. The users expressed satisfaction with the explanations produced, although one of the subjects commented that scripts sometimes had to be executed to understand exactly the effect of a particular line of code.

The fact that all users managed to produce successful scripts within four hours indicates that it is not excessively difficult to use. That the overall idea was easily understood and the input language quickly learnt suggests that this kind of system could minimise the overheads associated with the task of automated explanation generation for systems more complex than those illustrated here.

6 Related Work

BabyTalk BT-45 (Reiter et al., 2008) generates textual summaries of low-level clinical data from a Neonatal Intensive Care Unit. Summaries are created for different audiences, such as doctors and nurses, to help them in making treatment decisions. Generated summaries were found to be useful, but lacking in narrative structure compared to those created by humans. Further investigation is needed to determine where the trade-offs lie between acceptable explanations, underlying data complexity, and computational efficiency. Power (Power, 2012) describes OWL-Simplified English, a CNL for non-specialists used to edit semantic web ontologies in OWL, notably employing FS techniques for the definition of a user-oriented communication language. See also (Galley et al., 2001) whose NLG system combines FS) grammars with corpus-based language models. These works are limited to producing generators for any trace, rather than creating a higher-order framework which is used to write scripts which produce the generators.

7 Conclusions

Understanding why a violation occurred has many benefits for end-users of verification techniques and can save time when designing complex systems. The solution presented has the advantage of not being difficult to use by people with a computer science background, and can generate natural, easily understandable explanations despite inherent limitations of FS technologies. Should the constraints of regular languages prove to be such that this system would not be applicable in many areas, there is the possibility of *not* using FS techniques without any major changes in the framework's general architecture. Another possibility would be examining how the techniques discussed could be applied to provide dynamic explanations of online systems.

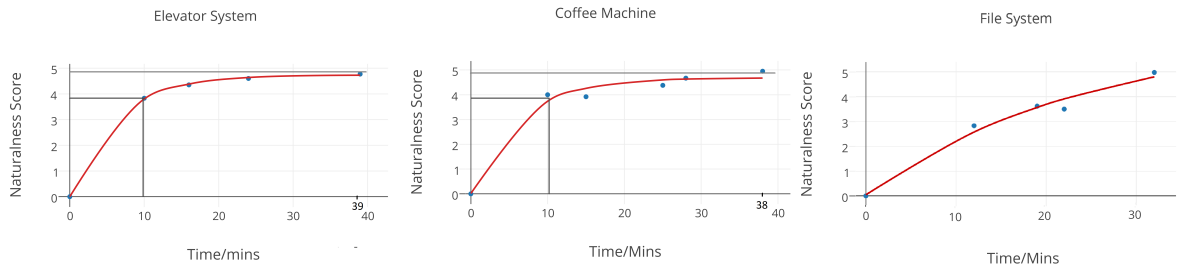

Figure 2: Graphs of the naturalness score given against the time after which the corresponding input script was created

References

Kenneth R Beesley and Lauri Karttunen. 2003. *Finite-state morphology*, volume 3 of *Studies in computational linguistics*. CSLI Publications.

Hercules Dalianis. 1999. Aggregation in naltural language generation. *Computational Intelligence*, 15(04).

Michel Galley, Eric Fosler-lussier, and Ros Potamianos. 2001. Hybrid natural language generation for spoken dialogue systems. In *In Proceedings of the 7th European Conference on Speech Communication and Technology*, pages 1735–1738.

Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys*, 28:196.

Tobias Kuhn. 2014. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, March.

Gordon J. Pace and Michael Rosner. 2014. Explaining violation traces with finite state natural language generation models. In Brian Davis, Kaarel Kaljurand, and Tobias Kuhn, editors, *Controlled Natural Language*, volume 8625 of *Lecture Notes in Computer Science*, pages 179–189. Springer International Publishing.

Richard Power. 2012. Owl simplified english: A finite-state language for ontology editing. In Tobias Kuhn and Norbert E. Fuchs, editors, *Controlled Natural Language*, volume 7427 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin Heidelberg.

Ehud Reiter, Albert Gatt, François Portet, and Marian van der Meulen. 2008. The importance of narrative and other lessons from an evaluation of an nlg system that summarises clinical data. In *Proceedings of the Fifth International Natural Language Generation Conference*, INLG '08, pages 147–156, Stroudsburg, PA, USA. Association for Computational Linguistics.

Shuly Wintner. 2008. Strengths and weaknesses of finite-state technology: a case study in morphological grammar development. *Natural Language Engineering*, 14(04):457–469.