TECHNICAL REPORT

Report No. CS2015-01
Date: August 2015

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

Rachel Farrell
Gordon J. Pace
Michael Rosner

Department of Computer Science
University of Malta
Msida MSD 06
MALTA

Tel: +356-2340 2519
Fax: +356-2132 0539
http://www.cs.um.edu.mt

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

Rachel Farrell
University of Malta, Malta.

Gordon J. Pace
University of Malta, Malta
gordon.pace@um.edu.mt

Michael Rosner
University of Malta, Malta
michael.rosner@um.edu.mt

Abstract: *The need for understanding what has lead to a failure in a computer system is crucial for addressing problems with such systems. In this report 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 technology which is finite-state. Results of a two-pronged evaluation dealing with naturalness and ease of use are described.*

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

Rachel Farrell
University of Malta, Malta.

Gordon J. Pace
University of Malta, Malta
gordon.pace@um.edu.mt

Michael Rosner
University of Malta, Malta
michael.rosner@um.edu.mt

Abstract: *The need for understanding what has lead to a failure in a computer system is crucial for addressing problems with such systems. In this report 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 technology which is finite-state. 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. Automated program analysis techniques are highly effective in this regard, but the information they produce is not necessarily understandable at all levels of the design chain. Whilst debugging and tracing tools such as scenario replaying yield system diagnostics that are understandable to developers, it is largely opaque to system designers and architects who typically require higher level, less technical explanations of certain carefully identified classes of program behaviour.

This is particularly the case in situations in which domain (but not necessarily technical) experts are involved in scripting parts of the system, typically using domain-specific languages [Hud96] or controlled-natural languages [Kuh14]. For instance, consider a scenario in which a fraud expert is responsible for specifying suspicious user behaviour in a financial transaction system. Users of the system violating the rules set up by the expert would trigger alerts to be seen by the fraud-handling team or by the fraud expert himself or herself. However, the user behaviour, if displayed

"literally", would typically contain information that is irrelevant or long-winded. To be more effective, irrelevant information should be filtered out whilst long-winded low-level traces should be summarised. Unlike developers, for whom a long list of system events is acceptable (and arguably ideal) since it can be explored using debugging tools, for these less-technical persons the user actions (and related system reactions) would be more effectively visualised and cognitively absorbed if presented in terms of a narrative of the relevant events.

In practice, system behaviour analysis tools, be they unit tests, runtime monitors, model checkers or system and user classification tools, typically produce system diagnostics as traces of system and user events. The problem boils down to a natural language generation 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. For instance, while explaining a system trace which violated a property which states that *'Dormant users should not be allowed to perform transfers'*, the behaviour of users other than the one whose behaviour violated the property (or possibly the one with whom that user interacted in his or her violating transaction) can be safely left out, as can that user's interaction with the advertising panel of the application. On the other hand, when explaining a violation of a property which states that *'The system should never deposit money from an advertising account unless a user activates related adverts'*, the users' interaction with the advertising panel is crucial to understand why a particular system history did not proceed as specified in the requirement. Similarly, the reason behind the violation of each property would be different, depending on what went wrong, why, whose fault it was, etc.

In [PR14], Pace and Rosner presented an approach in which they showed how a finite-state 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 work described is highly manual and would not be very feasible for someone like a quality assurance engineer to write. In this paper we show how their approach can be generalised to be applicable to explain violations of general specifications. Since, as we have argued, 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 would be responsible for the top half 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. In addition, using our approach, another artefact would be produced, the explanation specification, which embodies the domain-specific natural language

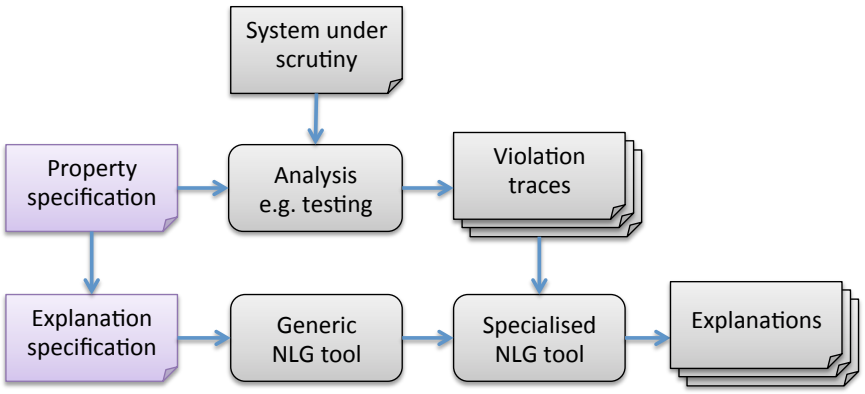

Figure 1: The architecture for general system diagnostics

information for the property in question. From this, a generic natural language generation tool will produce a specialised generation tool (embodying the domain-specific information and general information implicit in the traces) which can produce explanations for violations of the property in question. Our techniques have been implemented in a generic natural language generation tool, for which we show that the cost of adding user explanations for a property at an appropriate level of abstraction whilst ensuring naturalness of the text, 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).

Besides generalising the approach developed earlier, the present paper also further substantiates the claim that there is a place for finite-state methods in natural language generation.

The paper is organised as follows. Section 2 illustrates techniques that can be applied to yield progressively more natural styles of explanation. Section 3 details the "generator-generator" architecture used to generalise the previous approach. Section 4 explains the specification meta-language that we have implemented. Section 5 describes our evaluation and main results. Section 6 mentions some related work. Finally, some conclusions are drawn in section 7.

2 Trace Explanation Styles

Natural language generation [RD00] is typically seen as a multi-stage process, starting from document planning (defining content and structure), to microplanning (choosing words and means of expressing content) and surface realisation (creating the final

readable text). While one may choose to adopt the full English language as the target for the natural language generation of explanations of violation traces, we chose to adopt a controlled natural language [Kuh14] approach, in which have natural elements are restricted in the areas of syntax, semantics and the lexicon used. The target language for such specialised explanations typically consists of (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 [PR14], 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 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>`. We think of this trace as the concrete language which expresses the content we would like to express.

The simplest explanation, portraying the trace in the most basic way is achieved using the simple controlled natural language CNL0, in which 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.

An improvement to CNL0 is to add structure to the output which serves to improve the readability of the text. 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 [Dal99] techniques are used to 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.

It may be the case that the explanation contains detail which may be unnecessary or can be expressed in a more concise manner. CNL3 uses summarisation to achieve this — 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. 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.

Pace and Rosner [PR14] explored the use of finite state techniques to achieve a natural language explanation of a violation trace. The explanation language is considered as a CNL, whose basis, which includes how system actions present in the trace should be expressed, was described in XFST by a human author. The natural language explanation is obtained after having passed through different phases, each one using finite state technologies. Although finite-state technologies (such as automata and transducers) provide limited computational expressiveness, they have been used in the area of natural language processing and have been found to be well-suited for certain kinds of representations of linguistic knowledge, especially in the area of morphological parsing [Win08]. A variety of finite-state technology toolkits have thus been created, among which are HFST [LSP09], SFST [Sch06] and the toolkit we shall be using: XFST [BK03]. The Xerox Finite State Toolkit (XFST) is a toolkit used to build finite-state automata and transducers [Ran98], allowing for regular relations, which are sets of pairs of strings which can be compiled into a finite-state transducer. Starting from simply substituting trace symbols with sentences, every subsequent explanation becomes more natural, through the use of linguistic techniques such as structuring the text into paragraphs, aggregation, contextuality — as increasingly adopted in the CNLs above.

3 Generalised Explanations

Given a particular property, one can design a NLG tool specialised for that property, and capable of explaining violation traces of that property. However, it requires a substantial amount of work to write a generator for each property. In addition, some of the explanation improvements presented in the previous section, going from CNL0 to CNL3 are common to all, or at least most, properties. We thus chose to address the more general problem of trace violation explanations, in a manner 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).

In order to explain a trace emitted by a particular system, a specification of what different symbols occurring in it mean is required. Symbols can be explained in different ways: separately, in a particular context, and grouped together, depending on the type of output required; more groupings can possibly mean more summarised explanations. Using the gate example mentioned previously, we can explain `<o>` as: *The gates opened* or *The gates opened again* if it occurs after `<o,c>`. `<o,c>` can be explained as: *The gates opened and closed*. Also important is the explanation of errors, and how they come about. The contents of such explanations should include basic sentence elements like subject and predicate. Other information can be given which can enable us to know what kind of linking words to use (contrastive, sequential, additive, etc.) so that sentences can be aggregated, making their overall structure more fluid. Such a specification has to be given by a QA engineer, yet many of the linguistic details are likely to be outside such a user's immediate sphere of competence. For this reason a specification language was designed and implemented 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, which uses XFST [BK03], for the language of traces violating the property in question. In this manner, our approach is to go beyond a natural language generation system by developing a generator of trace explanation generators. Hence, our tool generates an XFST script using specifications given by the QA engineer. When the trace has been supplied, it is added to the script, which is then run by XFST. The explanation is then generated.

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 discussed below:

Explaining events: Individual events require to be explained to enable descriptions to be derived. Rather than give a complete sentence for each symbol, we split the information into the *subject* and *predicate*, thus allowing us to derive automat-

ically 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 `at` 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 prefer to 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, in the `EXPLAIN` section of the script allows for such terms to be explained more succinctly as can be seen in the example below:

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

Errors and assigning 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 how they should be structured into paragraphs. For example, using CNL1 as an example, we can add a newline after the lift arrives at a particular floor. Similarly, based on the example for CNL2, we specify that the sequence of events `<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) *natural sounding explanations:* Using our approach, what degree of naturalness can the trace explanations achieve? How much effort is required to achieve a high enough degree of naturalness in the explanations?; and (ii) *user acceptance*: How difficult is it for first time users to write specifications using our framework?

5.1 Effort In-Naturalness Out

Since, using our framework one can achieve a degree of naturalness 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. In order to measure this, we considered three different properties (one for an system controlling an elevator, one for a file system and one for 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 sequence of trace explanation scripts of increasing complexity, where the time taken to develop each script 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 being unnatural and difficult to follow, 6 being very natural and easy to follow[1]; the even number of different choices prevented the respondents from choosing a neutral score. Out of the total number of generated explanations, only a fraction were shown to each evaluator, and were presented in a random order rather than in the order they were generated. These measures were taken in order to prevent the human judges from making note of certain patterns, which might have incurred a bias. For the experiment we elicited over 477 responses from over 64 different persons.

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 explanations which fared best having a high rate of aggregation and summarisation. Interestingly, even with scripts written quickly e.g. 15–20 minutes[2] many evaluators found the explanations to be satisfactory with respect to naturalness.

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, which is close to 5 (corresponding to *fairly natural*). As marked on Figure 2, 80% of this limit is roughly achieved during the first 20–30% of the total time taken to

[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 such 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.

Table 1: Overall scores given to generated explanations

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

create the most comprehensive script we wrote[3]. The graph obtained for the file system traces gives a somewhat different view; a higher overall score is obtained than the other two graphs, yet we do not get the same initial steepness in gradient[4]. A possible reason for the discrepancy in the graph shape could be that traces obtained for this system all contained many repeated symbols in succession, and therefore, until the stage was reached when the script handled this repetition, the explanations received low scores. This shows that there may also be a relation between the kind of system we are considering and the effort and linguistic techniques required to generate natural sounding explanations for its traces.

From this experiment, one can conclude that whilst we can say that a certain inherent limits exist, natural-sounding explanations can be well achieved using this system. Effort however is rather important, and in most cases, the more time invested in building a script, the better the quality of the output obtained. Nevertheless, even with minimal effort, a script author highly trained in the input language can obtain a rather satisfactory output.

[3]This is reminiscent of the Pareto Principle, which states that for many phenomena 20% of the cause is responsible for 80% of the effect.

[4]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.

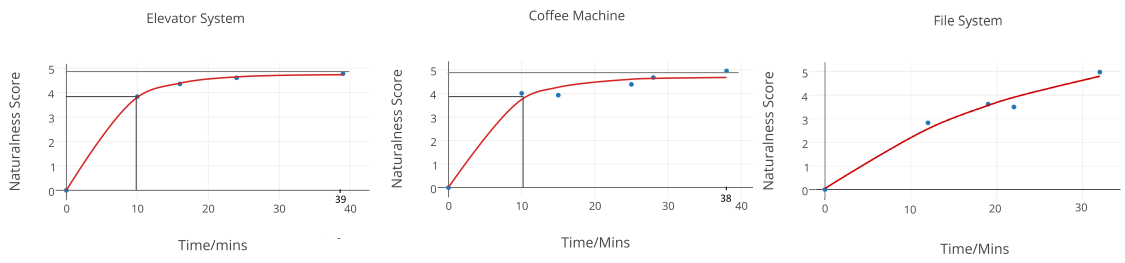

Figure 2: Graphs of the naturalness score given against the time after which the corresponding input script was created

5.2 User Acceptance Test

To assess how accessible the framework was for users, we ran a four-hour experiment with four users, who were already familiar with concepts such as regular expressions. These test-users were given a brief introduction to the system and its intended use, after which it they were requested to produce scripts to explain different properties without any further support. The users were then asked to comment about how easy they found the input language to use, whether they felt it restricted them from providing certain explanations, their levels of 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.

However, overall, these users characterised the scripting language between *somewhat difficult* to *easy to use*. Dealing with the contextual explanation of events presented the greatest challenges, although all managed to produce an error explanation which required the use of this concept. Apart from simply explaining every symbol as a complete sentence, the users also managed to create scripts involving aggregation and summarisation. When questioned, the users showed satisfaction with the explanations they managed to produce, 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 scripts that successfully achieved explanations within four hours of becoming acquainted with our system indicates that it is not excessively difficult to use. That the overall idea was easily understood and the input language quickly learnt suggests that a system of this kind could minimise the overheads associated with the task of automated explanation generation for systems more complex than those illustrated here.

6 Related Work

Although limited, there is other work from the literature attempting to generate explanations for different end-users, based on a sequence of actions or events. In most cases, however, the work is limited to producing generators for any trace, rather than a higher-order framework which is used to write scripts which produce the generators.

BabyTalk BT-45 [RGPvdM08] is a system which generates textual summaries of low-level clinical data from a Neonatal Intensive Care Unit created over a 45-minute interval. In our system, one may consider summarisation to make understanding an error easier for different professionals and similarly in this case, the summaries are created for different audiences, such as doctors and nurses, to help them in making treatment decisions. The automatically generated summaries were found to be useful, but somewhat lacking in narrative structure when compared to those created by humans. The fact that the generated explanations reported here were judged to be relatively natural is partly explained by the relative simplicity of our examples and the corresponding "stories" compared to the much more complex data handled by BabyTalk. Further investigation is clearly needed to determine where the tradeoffs lie between acceptable explanations, underlying data complexity, and computational efficiency.

Power [Pow12] describes OWL-Simplified English, a CNL used to edit semantic web ontologies in OWL. This work can be considered related to ours in that it seeks to empower users not familiar with the technical complexities of editing OWL by providing them with a purpose-built CNL for that job that is understandable to native English speakers. Power's work is also notable in that it is one of the few to successfully employ finite-state techniques for the definition of a user-oriented communication language despite the well-known fact that English is not a regular language.

7 Conclusions

Understanding why a violation occurred has many benefits for the end-users of verification techniques and can save precious amounts of time during the design phase of complex systems. Finite-state technology makes for a simple and efficient way of producing natural language but has its own limitations due to the restrictions of regular languages. However, we have found that our solution, which exploits the relative simplicity of finite-state specifications, has the advantage of not being difficult to use by people with a background in computer science, and has the possibility of generating natural and easily understandable explanations that might otherwise require familiarity with potentially complex linguistic issues.

An interesting possibility for future work would be to consider an system which provides a natural language explanation of actions performed by a system which is currently running, such as an online system. The techniques discussed in this work can be used to provide a dynamic explanation of the actions which happened up to the present.

If 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 finite-state techniques without any major changes in the general architecture of the framework.

References

[BK03] Kenneth R Beesley and Lauri Karttunen. *Finite-state morphology*, volume 3 of *Studies in computational linguistics*. CSLI Publications, 2003.

[Dal99] Hercules Dalianis. Aggregation in naltural language generation. *Computational Intelligence*, 15(04), 1999.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28:196, 1996.

[Kuh14] Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, March 2014.

[LSP09] Krister Linden, Miikka Silfverberg, and Tommi Pirinen. Hfst tools for morphology-an efficient open-source package for construction of morphological analyzers. In Cerstin Mahlow and Michael Piotrowski, editors, *State of the Art in Computational Morphology*, volume 41 of *Communications in Computer and Information Science*, pages 28–47. Springer Berlin Heidelberg, 2009.

[Pow12] Richard Power. 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, 2012.

[PR14] Gordon J. Pace and Michael Rosner. 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, 2014.

[Ran98] Aarne Ranta. A multilingual natural-language interface to regular expressions. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 79–90. Association for Computational Linguistics, 1998.

[RD00] Ehud Reiter and Robert Dale. *Building Natural Language Generation Systems*, volume 33 of *Studies in Natural Language Processing*. Cambridge University Press, Cambridge, UK, 2000.

[RGPvdM08] Ehud Reiter, Albert Gatt, François Portet, and Marian van der Meulen. 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, 2008. Association for Computational Linguistics.

[Sch06] Helmut Schmid. A programming language for finite state transducers. In Anssi Yli-Jyrä, Lauri Karttunen, and Juhani Karhumäki, editors, *Finite-State Methods and Natural Language Processing*, volume 4002 of *Lecture Notes in Computer Science*, pages 308–309. Springer Berlin Heidelberg, 2006.

[Win08] Shuly Wintner. Strengths and weaknesses of finite-state technology: a case study in morphological grammar development. *Natural Language Engineering*, 14(04):457–469, 2008.