Twelfth Annual Advances in Cognitive Systems Poster Collection (2025) 233-250

Generative Example-Driven Error Mining

Constantine Nakos CNAKOS@U.NORTHWESTERN.EDU
Walker Demel WALKER.DEMEL@NORTHWESTERN.EDU
Kenneth D. Forbus FORBUS@NORTHWESTERN.EDU
Qualitative Reasoning Group, Northwestern University, 2233 Tech Drive, Evanston, IL 60208 USA

Abstract

Hand-curated natural language (NL) systems use explicit linguistic knowledge to understand and generate text. However, maintaining this knowledge can be a challenge. Errors abound in linguistic resources of any size and tracking them down requires expertise that could better be applied elsewhere. In this work, we introduce Generative Example-Driven Error Mining (GEDEM), a technique for discovering errors in hand-curated NL systems capable of generation. GEDEM generates example sentences based on the system's linguistic knowledge and asks the user to identify any sentences that are ungrammatical or which differ semantically from the others. It then uses these judgments to diagnose the errors in the system's linguistic knowledge that caused the bugs in the generated examples. We show the utility of our framework by applying it to Companions Natural Language Understanding (CNLU), a knowledge-rich semantic parser with a generation component.

1. Introduction

Hand-curated natural language (NL) systems use explicit linguistic knowledge to understand and generate text. The use of explicit knowledge makes hand-curated systems more *transparent* and *scrutable* (Tintarev & Masthoff, 2007) than their counterparts based purely on machine learning (ML), but maintaining this knowledge can be a challenge. Errors abound in linguistic resources of any size, and finding them is complicated by the breadth of natural language, the subtlety of potential errors, and the expertise needed to engage with the specialized kinds of knowledge (e.g., grammar rules) used by hand-curated NL systems.

Prior research has made inroads into this problem. Our framework for Interactive Natural Language Debugging (INLD; Nakos & Forbus, 2024; Nakos, Kuthalam, & Forbus, 2022) lays out how debugging a hand-curated NL system can be cast as a reasoning problem, allowing non-experts to help debug the system by answering simple questions in natural language. However, INLD applies to sentences that are already suspected to contain an error. It does not address the problem of finding errors in the first place. Thus, additional techniques are required to help identify errors at scale.

Error mining offers a complementary approach. It is a family of techniques designed to find errors in the behavior of a syntactic parser. Typically, this is achieved by running the parser over a text corpus and identifying *n*-grams (de Kok & van Noord, 2017; Sagot & de la Clergerie, 2006; van Noord, 2004) that are commonly associated with fragmented parses. Similarly, Egad

(Goodman & Bond, 2009) uses round-trip parsing and generation to identify erroneous grammar rules. However, error mining for semantic parsers poses additional challenges. Error mining requires a signal to determine when the parser gets something wrong. The signal is used to assign *suspicion* to the *n*-grams or rules involved in the sentence, guiding the developers of the system to errors that can be fixed. Typically, the error signal is whether the sentence can be parsed or, in the case of Egad, whether it can be parsed, regenerated, or paraphrased.

But for errors in semantic knowledge, the signal is not so obvious. The system might be able to parse a sentence without interpreting it correctly, and taken by itself, the ability to regenerate or paraphrase the sentence says nothing about accuracy of the interpretation. One option is to use a semantically annotated corpus, such as OntoNotes (Hovy et al., 2006). The sentences can be parsed, and the interpretations can be checked against the provided annotations to produce an error signal. But such corpora are hard to come by, costly to produce, and tied to semantic representation schemes that a particular parser may not use. Furthermore, the annotations in a corpus are fixed. They do not reflect any changes to the ontology or representation language that may have proven necessary after the annotations were made.

We propose an alternative approach inspired by the work of Mittal and Moore (1996), which revealed errors in a knowledge base by sampling instances of the concepts it contained. Their key insight was that errors in concrete examples are easier to detect than errors in abstract knowledge, and that corner cases and unintended consequences can be exposed by sampling such examples. To debug a specification for floating point numbers, they generate concrete strings that are entailed by the specification. For instance, the rule "*floating-point-number* ::= [*sign*] {*digit*}* *decimal-point* {*digit*}* [*exponent*]" generates the string ".", which is not a valid floating point number. Spotting the problem with "." is easier and requires less expertise than finding the error in the underlying rule (i.e., the second * operator should be a +, ensuring at least one digit appears after the decimal).

We apply the same principle to language to address the difficulties of semantic error mining. Intuitively, grammatical and semantic errors are easier to spot in concrete sentences (e.g., the error of subject-verb agreement in "the ducks *flies*") than they are in linguistic knowledge (e.g., a typo buried in the agreement feature of a lexicon entry, an overly permissive grammar rule, etc.). By generating concrete examples that reflect the behavior of a system's linguistic knowledge, we can mine for potential errors without requiring a corpus of semantic annotations.

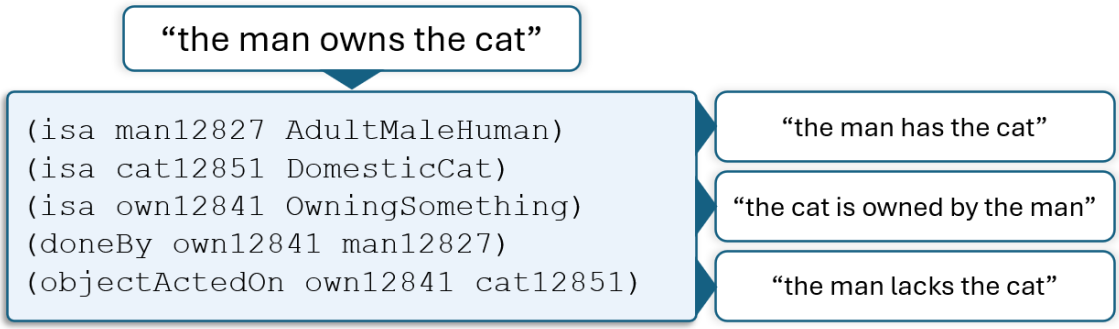

Figure 1. Semantics for "The man owns the cat," and sentences that CNLU parses to the same semantics. Note that the use of "lacks" here is due to an erroneous semtrans.

Thus, we introduce *Generative Example-Driven Error Mining* (GEDEM), a framework for semi-automatically finding errors in semantics-enabled hand-curated NL systems. GEDEM samples interpretations from the space of possible semantics produced by a parser, generates multiple sentences for the same interpretation, and uses *paraphrase and grammaticality judgments* elicited from a user as an error signal. Because the semantics for each sentence in the cluster are intended to be the same, any sentence that is not a meaning-preserving paraphrase of the others indicates an error in the linguistic knowledge used to generate the buggy sentence. The same goes for any sentences that are ungrammatical. Judging clusters of sentences for bad paraphrases and grammatical errors provides an error signal that can be used to flag whichever linguistic facts were used to generate the incorrect sentences.

An example can be seen in Figure 1. The left side of the figure shows the semantics for the sentence "The man owns the cat" produced by the CNLU semantic parser (discussed below). The right side of the figure shows three other sentences that yield the same semantics. The first two are good paraphrases of the original sentence, but the third, "The man lacks the cat," is not. This indicates an error in CNLU's semantic mappings: the verb "lack" maps to the ontological concept `OwningSomething`, so CNLU treats it as a synonym for "owns" or "has".

GEDEM has several useful properties. First, it begins by sampling semantic interpretations to use for generation, factoring out the problem of disambiguation that arises when parsing sentences from a corpus. GEDEM starts with fixed semantics that the generated examples should convey, rather than a piece of wild text for the system to paraphrase, so any incorrect examples reflect errors in the system's explicit linguistic knowledge, rather than failures of disambiguation. For example, paraphrasing "Male deer have antlers" as "Male deer eat antlers" reflects a disambiguation error rather than an error in explicit linguistic knowledge. GEDEM avoids confounding the two types of errors by sampling its own interpretations rather than attempting to parse and disambiguate text.

Second, generating multiple sentences at once helps assign suspicion (described in Section 3.3). Because two errors rarely cancel out, it is reasonable to assume that the linguistic facts (grammar rules, lexicon entries, semantic mappings, etc.) used to generate a correct sentence are error-free. Thus, when assigning suspicion for incorrect sentences, GEDEM can rule out the facts that were used to generate correct sentences in the same cluster. For example, in Figure 1, GEDEM would assign suspicion to the facts involved in generating the incorrect sentence "The man lacks the cat" that weren't also used to generate the other two sentences. That is, the mapping from "lack" to `OwningSomething`, which is only used by the incorrect sentence, is more likely to be erroneous than the rule combining the verb and the direct object, which is also used by the correct sentence "the man has the cat".

Third, even though the GEDEM framework has a human in the loop, the task is a simple one that requires no specific expertise, making it a prime candidate for automation with a large language model (LLM). In Section 5, we conduct a preliminary experiment to test the feasibility of this approach, showing tolerable agreement with human judges, with plenty of room for improvement via few-shot prompting or use of a more powerful model. If an LLM can replace the human judges, GEDEM becomes a fully automatic technique for flagging potential errors in a system's linguistic knowledge, subject to the limits of the sampling method and the system's generation capabilities.

Finally, while the intention of GEDEM is to uncover errors associated with bad paraphrases, the example generation process itself serves as a form of error mining. Failures while generating

examples can reveal important limitations of the NL system, as we observe in Section 5. Disparities between parsing and generation, failure to generate sentences with certain grammatical constructs, or other, similar failures can provide grist for debugging, even in the absence of paraphrases that can be judged. The breakdown of errors between the earlier stages of the pipeline and full examples passed to the user for judgment will depend on the reliability of the system's generation capabilities, the method chosen to sample semantic interpretations, and whether GEDEM is using knowledge that has been previously debugged or is just being tested for the first time.

We ground our discussion of GEDEM in Companions Natural Language Understanding (CNLU; Tomai & Forbus, 2009), the natural language component of the Companion cognitive architecture (Forbus & Hinrichs, 2017). CNLU is a broad-coverage semantic parser designed as an input modality for learning and reasoning. Its reliance on hundreds of grammar rules, tens of thousands of semantic mappings, and hundreds of thousands of lexicon entries makes it a perfect test bed for GEDEM, which can flag potential errors without requiring a CNLU expert to crawl through the system's linguistic knowledge by hand.

The remainder of the paper is as follows. In Section 2, we lay out necessary background for our work, focusing on the basic concepts and capabilities of CNLU. In Section 3, we discuss the GEDEM framework in more detail, including the *generative story* used to sample interpretations, how examples are generated and judged, and how linguistic facts are assigned a *suspicion score* based on user error judgments. In Section 4, we present *say*, a new Natural Language Generation (NLG) component within CNLU, and discuss how it can be used with GEDEM to find errors in CNLU's linguistic knowledge. In Section 5, we show the efficacy of GEDEM by evaluating our implemented system on a set of generated examples and discussing some of the errors we found in the process. Finally, we review related work in Section 6 and conclude in Section 7.

2. Background

One of the primary goals of the Companion cognitive architecture (Forbus & Hinrichs, 2017) is to create *software social organisms*, software systems that can reason, learn, and interact the way that people do, as a step on the path to true Artificial Intelligence. In order to achieve this goal, the Companion architecture is equipped with input modalities, such as language and sketching (Forbus et al., 2011), that allow the system to learn from examples and collaborate with its human partners.

Language is handled by CNLU (Tomai & Forbus, 2009), a broad-coverage semantic parser that takes text as input and outputs CycL predicate calculus (Lenat & Guha, 1991) grounded in the NextKB ontology[1] and suitable for downstream reasoning. CNLU uses a rule-based, bottom-up chart parsing algorithm based on the TRAINS parser (Allen, 1994), along with a context-free, feature-based grammar. The parsing algorithm has been extended to handle semantics as well as syntax. As parse trees are built up, the semantics of their constituents are extended and combined to produce the final interpretation of the sentence.

CNLU depends on several types of hand-curated linguistic knowledge stored in NextKB for language understanding. *Lexicon entries* map the surface form of a word to its root form, part of speech, and grammatical features. Input tokens are looked up in the lexicon, and the relevant lexicon entries are converted into leaf-level syntactic *constituents* that the parser builds up into

[1] https://www.qrg.northwestern.edu/nextkb/index.html

parse trees. *Grammar rules* control how smaller constituents combine into larger ones, such as a determiner and a noun combining to form a noun phrase (e.g., "the" + "cat") or a noun phrase and a verb phrase combining to form a clause (e.g., "the man" + "owns the cat"). Grammatical features enforce constraints such as subject-verb agreement. They also govern how the semantics of the child constituents are combined, and they bind grammatical functions (e.g., subject or object) so the semantics can be updated accordingly.

The semantics of a word are instantiated from *semtranses* ("semantic translations"), mappings from the root form of a word to a semantic template representing its meaning. As the semantics are passed up the parse tree, the template is filled in with *discourse variables*, representing entities mentioned in the text, as they are bound to the appropriate grammatical functions. The end result is one or more fully instantiated predicate calculus statements that represent one of the word's possible meanings. Verbs are represented as Neo-Davidsonian events (Parsons, 1990), reified entities connected to their participants with *role relations*.

Valence patterns encode the different syntactic and semantic arguments a verb can take. When grammar rules bind grammatical functions (e.g., `:SUBJECT`) to discourse variables (e.g., `man4201`), the appropriate role relations are instantiated to link the event (e.g., `own4202`) to its participant, adding a statement like `(performedBy own4202 man4201)` to the verb's interpretation. CNLU's valence patterns were mostly extracted from the FrameNet corpus (Fillmore, Wooters, & Baker, 2001) using a noisy, semi-automatic process. This makes them a prime target for error mining.

CNLU's approach to language understanding has advantages and disadvantages. On the one hand, its use of explicit linguistic knowledge means that the system can be surgically debugged and extended. This is a boon in cognitive systems research, where finding the right way to represent a piece of knowledge is an important part of the process. Changes in representation, the discovery of errors, or gaps in coverage can all be addressed directly with a few tweaks to the appropriate linguistic facts.

On the other hand, maintaining the system requires a fair bit of effort. CNLU has 191,000 lexicon entries, 70,000 semtranses, and 467 grammar rules. Many of these facts were derived semi-automatically from other resources, while others were authored by hand. While CNLU has a suite of grammar regression tests that shed some light on its performance, they do not cover every piece of linguistic knowledge it has. Put simply, CNLU has errors in its knowledge that are hard to find and require expertise to fix by hand. While INLD (Nakos & Forbus, 2024; Nakos, Kuthalam, & Forbus, 2022) reduces the expertise needed to debug errors once they have been found, proactively finding errors at scale remains an open problem, motivating our creation of GEDEM.

So far, Companions' use of language has mainly focused on understanding. When generation is needed to communicate knowledge to the user, as is the case in the information kiosk described by Wilson et al. (2019) or its online version (Nakos & Forbus, 2025), Companions use *verbalize*, a template-based NLG system that converts predicate calculus into English text. However, the generation needs of GEDEM are more extensive and more particular. In order to reveal errors in CNLU's linguistic knowledge, GEDEM requires an NLG system that uses the same knowledge, as opposed to the separate set of canned strings and templates used by *verbalize*. Likewise, we want Companions to be able to express knowledge using flexible, grammatically aware text generation rather than stilted text generated by filling in templates.

To address these shortcomings, we introduce *say* (Section 4), an NLG system that generates sentences from CNLU-style semantic representations using CNLU's linguistic knowledge. Using *say* with GEDEM, we can search for errors in CNLU's knowledge by identifying faulty generated sentences. Finding and correcting these errors will help prepare *say* for future applications, where we hope it can take the place of *verbalize* in communicating knowledge to a user via text.

Next, we turn our attention to the GEDEM framework itself.

3. The GEDEM Framework

GEDEM works by generating a cluster of example sentences with a semantic parser from the space of possible interpretations for a given sentence. For each interpretation, a user judges which examples are ungrammatical and/or bad paraphrases of the others. Performed over enough clusters, this provides an error signal that can be used to identify likely errors in a system's linguistic knowledge. The list of errors can then be given to an expert to be confirmed and corrected. This framework is shown in Figure 2.

Before discussing the specifics of GEDEM, it is worth mentioning two of its limitations. The first is that the approach requires an NLG system that taps into the same linguistic knowledge as the parser, even if only a crude one. The need for a generation system to complement the parser is one of our motivations for the creation of *say*, the NLG system presented in Section 4.

The second limitation is that GEDEM focuses on *errors of permission* rather than *errors of restriction* (Nakos & Forbus, 2024). The former occur when a system's linguistic knowledge is too permissive, accepting ungrammatical sentences or producing nonsensical interpretations. The latter occur when the linguistic knowledge is too restrictive, rejecting grammatical sentences or failing to find desired interpretations. Errors of permission are most visible during generation, when a bad sentence is generated, while errors of restriction are most visible during parsing, when the right interpretation can't be found. GEDEM deals with the former case. From a parsing perspective, GEDEM cleans up the system's linguistic knowledge and decreases the number of incorrect interpretations considered by the parser. From a generation perspective, GEDEM keeps the system from producing ungrammatical nonsense. We leave the inverse cases for future work.

In the remainder of this section, we lay out the GEDEM framework in detail, along with our implementation of it for CNLU. In Section 3.1, we describe the generative story GEDEM uses to sample interpretations. In Section 3.2, we discuss how examples are generated and scored. Lastly, in Section 3.3, we show how potential errors are scored.

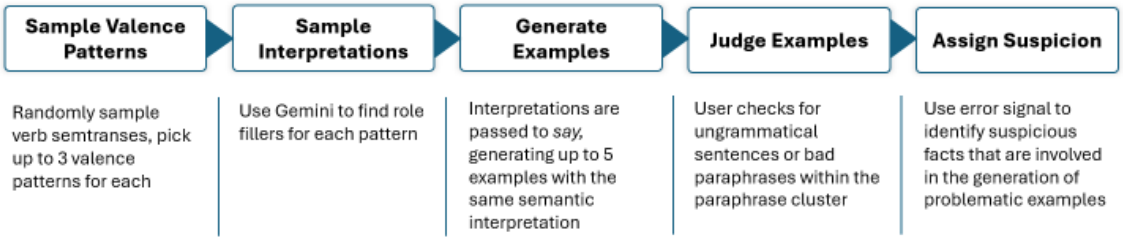

Figure 2. The GEDEM pipeline.

3.1 The Generative Story

One of the core components of GEDEM is the *generative story*[2] that samples semantics from the space of possible interpretations. This component is flexible. Any random function that generates valid semantics will suffice, although the nature of the function will determine what pieces of knowledge GEDEM can test. For example, a generative story that only produces semantics for simple declarative sentences will leave the system's rules for generating questions untested. When selecting a generative story for GEDEM, there is a tradeoff between breadth and depth. A powerful generative story that samples from the full space of the system's semantics will, at least in principle, allow GEDEM to test the entirety of the system's linguistic knowledge. However, a simpler and more restricted generative story can better target specific types of knowledge. If the developer's current objective is to debug simple declarative sentences, generating questions is just a distraction.

For CNLU, the space of possible interpretations is large, so the generative story can be quite complex. On top of basic semantics for nouns and verbs, CNLU handles quantifiers, modals, relative clauses, conjunctions, and more. This representational richness is to be expected in a broad-coverage semantic parser, but it raises the question of where to even begin debugging.

We opt for a simple generative story designed to mine for errors where they most need to be found: semtranses and valence patterns. Semtranses and valence patterns are vital for producing interpretations with CNLU, they number in the thousands, and most of them were produced semi-automatically from FrameNet (Fillmore et al., 2001) and have not been inspected by hand. They are also similar enough to one another that the generative story does not have to be complicated to produce useful results.

Our generative story is as follows: Sample a verb semtrans at random, sample one of its valence patterns, and then instantiate its role relations with *role fillers*, entities that play a specific role in a typical event of that type. The result is a semantic expression describing a single Neo-Davidsonian event and its participants, corresponding to a simple declarative sentence such as "The man owns the cat" or "The girl gave the boy a book."

Role fillers are an additional type of knowledge the system needs so it can generate plausible examples. While role relations in NextKB have type constraints that limit what arguments they *can* take (e.g., `performedBy` can only take an `Agent-Generic` as its second argument), NextKB does not have information about what arguments would be *typical* or *exemplary*. Furthermore, the same role relations are used for different types of events where the typical fillers might vary. For example, `instrument-Generic` appears as a role relation in valence patterns for over 200 verbs, but the instruments typically used for, say, "baptize" and "pulverize" are very different.

To fill in this missing knowledge, we elicit role fillers from an LLM with fill-in-the-blanks questions, as shown in Figure 3. Whenever a valence pattern is sampled for GEDEM, if the knowledge base does not already contain an example of that valence pattern, we query Google Gemini 2.5 Flash[3] with the prompt shown in Figure 3 in a zero-shot context with thinking disabled.

---

[2] We use this term in the Bayesian sense: an account of how a random process yields values for an observed variable. In the case of GEDEM, the observed variable is a semantic interpretation suitable for generation and the random process is a method for sampling it from the system's linguistic knowledge.

[3] https://ai.google.dev/gemini-api/docs/models#gemini-2.5-flash

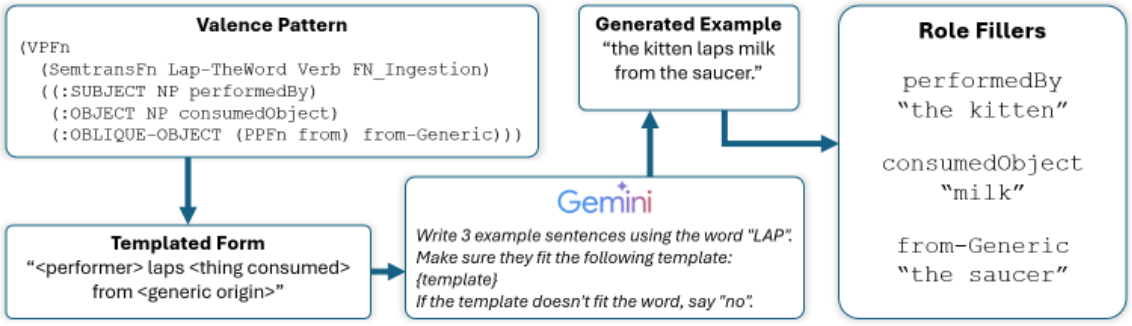

Figure 3. After sampling a valence pattern for a semtrans, we automatically convert to a templated form, which is sent to Gemini as part of a prompt asking for filled-in sentences. Gemini sends back a maximum of 5 examples sentences which are used to get role fillers.

The fill-in-the-blanks templates are produced automatically using simple ordering heuristics (e.g., the subject goes before the verb and the object goes after), and the role hints are generated depending on the role relation expected in that position.[4] The examples given by the LLM are parsed and regenerated to ensure they actually yield the desired role fillers, and then they are stored in the knowledge base for future use. If the LLM fails to generate an example for a valence pattern, or if the example fails to parse or regenerate as intended, this is an early signal that the valence pattern may be incorrect or that the example poses a challenge for CNLU and *say*.

Once an example for a valence pattern has been found, the role fillers from that example are used to instantiate a complete semantic expression for the sampled verb semtrans. Discourse variables are instantiated for the event and its participants, role relations are added to the basic semantics for the verb, and the semantics of the role fillers are added as well. The end result is a semantic interpretation that could have been produced by CNLU, which serves as the basis for a cluster of generated example sentences.[5]

3.2 Generating and Judging Examples

Once the semantics have been sampled, the next step is to generate a cluster of example sentences that express those semantics in various ways. All of the examples in a cluster should produce the seed interpretation when parsed, making them meaning-preserving paraphrases of one another. Thus, any discrepancies in meaning between them indicate that there is an error in the linguistic knowledge used to generate the outlier sentence. The same goes for grammatical errors.

[4] It is possible that the LLM does not interpret the word in the way it is intended, providing an example of how the valence pattern would apply to a *different* sense of the word. Currently, this is a source of noise in our system, introducing a potential mismatch between the semantics of a verb and its role fillers. In future work, we will extend the prompt with the word's intended meaning, akin to the clarification strings used to help the user distinguish between word senses in INLD.

[5] Another way to view this generative story is that we are sampling a verb semtrans and valence pattern, eliciting an example sentence for that valence pattern, then parsing it to produce the interpretation that GEDEM will use to generate example sentences. However, in future versions of the generative story, the semantics may be stitched together from cached role fillers, rather than being derived from a specific sentence.

```
The following sentences are all supposed to mean the same thing.
Which of them doesn't mean the same thing or is ungrammatical?

 1) The horse tramples through the mud
 2) The horse has trampled through the mud
 3) The horse is trampled through the mud

(Enter a list of numbers between 1 and 3, "none", :help, or :exit.)
> 3
```

Figure 4. A text interface that asks a user about grammaticality and paraphrase quality.

The errors found this way are not necessarily related to the linguistic knowledge used during the generative story. For example, the verb semtrans, valence pattern, and role fillers that an interpretation was sampled from could all be perfectly fine, yet the generated sentences may still be buggy. This occurs when the generated sentence uses a different semtrans or valence pattern to convey the same semantics, when a lexicon entry is incorrect, or when one of the sentences is generated using an incorrect grammar rule.

Thus, although the generative story only looks at verb semtranses and valence patterns, it can uncover errors in all kinds of linguistic knowledge, depending on how far afield the NLG system is allowed to go while generating the sentences. The advantage of this is that GEDEM does not have to be looking for anything in particular in order to find errors. Just running the process on new examples will uncover new errors to fix, subject to the coverage of the generative story and the constraints on text generation.

The CNLU implementation of GEDEM uses *say* (described in Section 4) to generate sentences from semantics. Once the clusters of examples have been generated, the system elicits error judgments by asking the user multiple choice questions via a simple text interface, as seen in Figure 4. The system keeps track of the linguistic facts used to generate each example so that suspicion can be assigned appropriately, as discussed next.

3.3 Scoring Linguistic Errors

The output of GEDEM is a list of linguistic facts that are likely to be erroneous, given the bad examples identified in the previous step. Following in the footsteps of prior error mining work, GEDEM calculates a *suspicion score* for each linguistic fact used in generating its examples. The suspicion score is a measure of how responsible the fact is for the observed bad examples, a guide for further debugging efforts.

There are two broad approaches to calculating suspicion scores in the error mining literature: a global score based on the ratio of the number of incorrect sentences that a feature appears in to the total number of sentences where it appears (van Noord, 2004), and an iterative score that assigns blame within each sentence, down-weighting features that just happen to co-occur with the true culprits (de Kok & van Noord, 2017; Sagot and de la Clergerie, 2006). We opt for the former approach, which is both simpler and a good fit for GEDEM's needs. Unlike traditional error mining, GEDEM generates clusters of sentences that can be compared to each other while assigning suspicion. Because of the high degree of overlap between generated sentences in a cluster, typically

only a few facts will differ between them. It is rarely the case that one error can cancel out another (e.g., "the ducks fly", where "ducks" is mistakenly a third-person *singular* noun and "fly" is mistakenly a present-tense third-person *singular* verb), so if a fact can be used to generate a correct sentence, it should be absolved of suspicion for that cluster.

Our GEDEM implementation assigns suspicion scores as follows. Let f_i be a linguistic fact, let s be a generated sentence, let C_s be the cluster that contains it, let $Corr(C_s)$ be the subset of correct sentences in C_s, and let $F(s)$ be the set of facts used to generate s. Next, define $Inc(s)$ as the subset of $F(s)$ that was not used to generate any correct sentence in C_s:

$$Inc(s) = F(s) \setminus \bigcup_{s' \in Corr(C_s)} F(s')$$

Note that $Inc(s)$ will be empty if s is correct. Then we can define the suspicion score $susp(f_i)$ for fact f_i as:

$$susp(f_i) = \frac{\sum_{s \in S} \mathbf{1}[f_i \in Inc(s)]}{\sum_{s \in S} \mathbf{1}[f_i \in F(s)]}$$

where $\mathbf{1}[\varphi]$ = 1 if φ is true and 0 if φ is false. In other words, suspicion is assigned to a fact whenever it is used to generate an incorrect sentence but not any of the sentences in the same cluster. The suspicion score for a fact is the ratio of sentences where this is the case to the total number of sentences where the fact is used. In other words, the suspicion score is a measure of how frequently a fact is used to generate incorrect sentences, factoring out instances where its use in a correct sentence in the same cluster absolves it of blame.

For example, in Figure 1, suppose "The man lacks the cat" is judged incorrect while "The man has the cat" is judged correct. Facts that are used to generate the former sentence but not the latter would be assigned suspicion. Thus, the mapping from "lack" to `OwningSomething` would be suspect, as would the lexicon entry for "lacks", which is used for the incorrect sentence but none of the others in the cluster. Note that GEDEM cannot tell which of the two facts is the real culprit from this cluster alone, but if the "lacks" lexicon entry appears in other clusters in sentences that are judged to be correct, its suspicion score will decrease, causing the spurious correlation to "wash out" as more examples are tested. However, taking full advantage of this property requires large datasets and targeted generation. In our current experiment, we treat the suspicion score as a coarse indicator of which facts *might* have errors, rather than trying to filter out false positives.

Next, we turn to *say*, the NLG system used in our implementation of GEDEM for CNLU.

4. Say

As discussed in Section 2, the Companion cognitive architecture has historically lacked a grammar-driven NLG system, instead relying on canned strings and templates to communicate knowledge to the user. The linguistic knowledge of CNLU is an obvious starting point for such a system. If the same lexicon entries, grammar rules, and semtranses currently used for understanding can also be used for generation, the system will come equipped with broad generation capabilities suited for use with a Companion's knowledge.

To that end, we introduce *say*, an NLG system that generates English phrases or sentences from CNLU-style semantic representations. *say* aims for *parity* with CNLU's semantic parsing. Ideally, any text it generates should produce the same interpretation when parsed by CNLU, if the

appropriate choices are selected. Likewise, if CNLU parses a sentence, *say* should be able to regenerate it, akin to the round-trip generation used by Goodman & Bond (2009). *say*'s input is a semantic expression like the one seen in Figure 1, and its output is one or more sentences conveying those semantics.

In practice, achieving parity poses several challenges. CNLU, its parsing algorithm, and its grammar were not designed with generation in mind, so care must be taken to capture the same space of possibilities without introducing discrepancies. Likewise, *say* should avoid introducing knowledge specific to generation whenever possible, both to reduce the maintenance burden and to ensure that understanding and generation can be debugged in tandem.[6]

With these considerations in mind, *say* takes a *generation by parsing* approach. Rather than attempting to reverse CNLU's parsing algorithm in all its complexity, including its specialized handling for grammatical function substitutions, valence pattern expansions, quantifier merging, and more, *say* uses the existing parsing code as much as possible. But instead of parsing a single sequence of input tokens, it parses a whole space of them at once, continuing until it finds a parse tree that captures the desired semantics. In essence, *say* winnows through a set of sentences that *might* yield the desired semantics when parsed by CNLU until it finds one that actually *does*. Various optimizations keep the computation tractable, and aggressive semantic filtering ensures that only sequences of tokens that could yield the input semantics are kept. Once *say* has found a parse tree with the right semantics, the output text can be decoded from the tokens at its leaves.

4.1 Setup

Generation by parsing requires bridging one key gap. Parsing has fixed tokens but unknown parse trees and semantics. Generation has fixed parse trees and semantics but unknown tokens. The *say* algorithm must both cover the desired semantics and account for variability in word choice and ordering.

To cover the desired semantics, *say* finds all semtranses that could cover at least one of the predicate calculus statements in the semantic representation. For example, if the input semantics contain `(isa own4202 OwningSomething)`, *say* will find all semtranses that match the template `(isa :ACTION OwningSomething)`. This includes the verbs "belong", "have", "lack",[7] "own", and "possess".

To handle variability in word choice, *say* tracks variable bindings (e.g., `:ACTION` must be bound to `own4202`), looks up the lexicon entries for these words, and instantiates constituents for parsing as if the words had been passed to CNLU as input. Semtranses with compatible parts of speech, features, and bindings are grouped into single constituents, letting one parse tree cover multiple possible sentences with shared syntax. Whereas the leaf constituents in CNLU represent a single token each, the leaf constituents in *say* represent multiple possible tokens with the same meaning, packing a large space of word choices into a manageable number of constituents.

[6] For future applications where fluency is important, using generation-specific heuristics or saving prior examples may be worthwhile. Such guidance would be analogous to the disambiguation system used when parsing. As our current focus is debugging, we leave questions of fluency and generation-specific guidance for future work.

[7] "lack" mapping to `OwningSomething` is an erroneous semtrans and one of our motivating examples.

Function words (e.g., determiners or prepositions) affect the grammatical structure of the sentence but may not have semtranses. To keep the number of parse trees from blowing up, *say* packs all the function words of a given type into a single placeholder constituent that represents the presence of a to-be-determined function word of that type. The specific word is chosen and checked later during decoding. Thus, content words guide the search for a parse tree, while function words are included where necessary to support the sentence's grammatical structure.

Unlike parsing, generation does not fix token ordering. Words under consideration can appear in any position, and *say* will only find valid orderings for them upon completing a parse tree. For example, when generating "The man owns the cat" and "The cat is owned by the man", *say* reuses the same constituent for "the man", even though the two sentences place it in different positions. As the syntax and the semantics for the phrase are the same in both cases, there is no need to derive it twice for the two different positions.

To reuse constituents in different positions, *say* requires a slight tweak to the chart used by the parsing algorithm. *say* uses a *circular chart* that only has one position, which it traverses repeatedly in both directions to form new constituents. Thus, "the man" could join forward with "owns the cat" or backwards with "by", and the resulting constituents would be stored back into the chart's single position for future use. This allows *say* to consider all word orderings at once without doing redundant work.

4.2 Generating Parse Trees

Apart from the changes needed to accommodate the circular chart, the core parsing algorithm is the same as CNLU's. That is, the *say* algorithm searches for sequences of constituents in its chart that match rules in its grammar. When a match is found, the rule is applied to produce a new constituent which is added to the chart. The new constituent can then be combined with others that are already in the chart, continuing the cycle until a complete parse tree is found. The code for matching rules, combining semantics, binding grammatical functions, managing choices, etc. is shared between *say* and CNLU. Reusing the code this way sidesteps the problem of having to reverse these operations for generation. It also supports parity between parsing and generation because any changes to the CNLU parsing algorithm will automatically be reflected in *say*.

The output of this step is one or more complete parse trees that cover the desired semantics, built up from the input constituents described in the previous section. The final decoding step will perform a check to handle irregularities related to specific word choices (e.g., rules that only apply to certain prepositions), so the use of a single constituent to cover multiple possible words is safe.

To keep parsing on track, *say* filters based on *semantic subsumption*. Any constituent whose semantics do not *subsume* the target semantics is dropped from consideration. This accounts for the various ways a constituent could introduce extraneous semantics, such as a new quantifier or role relation that isn't in the target semantics. Once the extraneous semantics are introduced, they cannot be removed, so there is no point keeping that constituent.

More broadly, the setup step ensures coverage of the target semantics by introducing all of the constituents the algorithm might need to assemble a complete parse tree with the correct semantics. The subsumption filter ensures that the constituents are assembled in the right way during parsing, pruning out dead ends where the algorithm has gone astray. Parsing continues until a constituent

has been found with semantics that subsume the target, meaning the two sets of semantics are equivalent.

4.3 Decoding Output

Once the parsing algorithm has produced one or more parse trees that cover the target semantics, the final step is *decoding* them into output strings. Each parse tree found by *say* encodes a space of possible sentences that share the same syntactic structure. These sentences can be generated by splitting out the individual word choices, reparsing them, and constraining the parse to the original parse tree. This step tests which combinations of words are compatible with the previously found parse tree and also produce the desired semantics. For example, a parse tree encoding "{the | this} {cat | feline}" would be broken into "the", "this", "cat", and "feline", then reparsed to find "the cat", "this cat", "the feline", and "this feline". When one such parse tree is found, the tokens are read from the leaf constituents and detokenized into a single string for output.

For GEDEM, we constrain *say* to reproduce role fillers as they were given. Since the main purpose of role fillers is to provide a concrete example of a valence pattern, there is no point in paraphrasing them during generation. Thus, if the user provides "the cat" as a role filler, *say* will never generate "the feline" instead. This constraint narrows the space of possible sentence variations, allowing *say* to focus on variations that are more interesting for GEDEM.

4.4 Limitations

say suffers from two key limitations: CNLU's grammar was not designed for generation, and the size of the search space can quickly become intractable. The former is just a question of grammar engineering. Rules that are infinitely recursive, overly permissive, or otherwise buggy should be fixed to bring *say* closer to parity with CNLU. Indeed, the purpose of GEDEM is to expose such errors so they can be fixed.

The latter problem is more nuanced. Due to placeholder words, the large number of grammar rules in CNLU, and the need to accommodate variable word ordering, the number of constituents grows rapidly. Currently, *say* uses a simple, hand-tuned scoring heuristic to prioritize constituents that have high semantic coverage, compact parse trees, and few open bindings. We also impose a 30-second timeout to halt runaway generations.

To further constrain generation, we restrict *say* to a whitelist of roughly 30 rules that handle some of the grammar's core functionality. Specifically, we allow the rules that bind arguments to verbs, convert verb phrases into complete sentences, and form prepositional phrases from noun phrases. These rules suffice to cover the bulk of CNLU's verb valence patterns, which in turn form the backbone of its understanding and generation capabilities. As we develop *say* further, we will extend this whitelist to cover other grammatical constructions, such as more complex noun phrases, conjunctions, and adverbs. In doing so, we can focus GEDEM on portions of the grammar where it is most likely to bear fruit.[8]

[8] Note that we also allow *say* to use the rules which are necessary to reproduce role fillers verbatim. Thus, even though the current whitelist excludes most of CNLU's rules for noun phrases, our GEDEM implementation can still use the role fillers that are pertinent to the sentences it generates.

5. Evaluation

As an initial evaluation, we ran our GEDEM implementation on 200 valence patterns sampled from CNLU's verb semtranses. The flow of data in our experiment is shown in Figure 5. For each valence pattern, we sampled three example sentences from Gemini, parsed them to find role fillers, and kept the first example where *say* could regenerate all of the role fillers. There were 87 valence patterns where at least one role filler from each example could not be regenerated, indicating gaps in the coverage of CNLU or *say*. We observe that *say* has trouble with modifiers, which limits which examples it can handle. Given that the sampled valence patterns have up to five arguments, each of which needs a role filler that's simple enough for *say* to regenerate, the large drop-off is to be expected. Tweaking the Gemini prompt could yield easier examples by encouraging the model to avoid modifiers, which would sidestep the problem for now. But the fillers that fail to regenerate constitute a promising testbed for future development, so we leave our setup as-is and save the failed examples for further debugging.

Out of the 200 valence patterns, 113 had at least one of their examples pass the test, forming a semantic interpretation. We then had *say* generate up to five GEDEM examples per interpretation. Of the 113 interpretations, 24 generated at least one example to be judged, with half of those reaching a full complement of five examples and all but two clusters containing at least two examples to compare. 89 interpretations were not able to generate an example within the allotted 30-second cutoff, indicating that *say* could not find a way to combine the role fillers into a valid sentence. This can be caused by missing rules in the whitelist, rules that need adjustment for use in generation, or an inadequate search heuristic. Unfortunately, these generation failures are not as clear-cut as the clusters containing incorrect sentences, but they do provide a useful starting point for improving *say*.

Finally, we gathered judgments for the clusters of generated examples. The first and second authors judged all 24 clusters of examples. Out of 94 total sentences, the raw agreement was 85%, and Fleiss's κ of 0.70. This is evidence that user judgments are broadly comparable for this task, indicating that the task is reasonable and paving the way for solo annotation of larger datasets.

As a preliminary test of whether GEDEM is suitable for automation, we queried Google Gemini 2.5 Flash using the question prompt in Figure 4 in a zero-shot context with thinking disabled and no prompt engineering. In a pairwise comparison with human annotators, the LLM achieved 67% raw agreement and Fleiss's κ of 0.54 in both cases. While lower than the agreement between humans, Gemini showed tolerable base case performance, indicating that modern LLMs may be up to the task of automating GEDEM. As *say* becomes more reliable, increasing the throughput of our pipeline, we intend to use few-shot prompting, prompt engineering, and Chain-of-Thought reasoning to optimize the model's performance.

5.1 Discussion

Our evaluation uncovered a variety of errors at different stages of the pipeline. We inspected a subset of the role fillers that failed to regenerate, the interpretations that failed to generate multiple examples, and the facts flagged as having high suspicion according to our error judgments. Some general trends are as follows:

- *Passive valence patterns* — Asking Gemini to generate sentences for valence patterns revealed a shortcoming: CNLU does not keep track of the difference between valence patterns used for

active sentences and ones used for passive sentences. This means that "The ball *is hit by* the boy." and the ungrammatical sentence "The ball *hits by* the boy." share the same semantics. While this is fine when parsing the former sentence, CNLU will treat some active sentences as passives, and *say* will produce paraphrases that don't respect the distinction. Fixing this issue will require both reworking the relevant grammar rules and correcting passive valence patterns.

- *Lack of synonyms* — *say* sometimes fails to generate more than one GEDEM example due to a lack of synonyms. For example, "whip" means `HittingSomethingAsIfByWhipping`, a NextKB collection shared by no other semtranses. Thus, *say* will never find a paraphrase for it. Likewise, valence pattern coverage is splotchy, and verbs that share a collection may not have any overlapping valence patterns to express the same sets of arguments. In most cases, this appears to be a function of coverage rather than differences between the verbs. Fleshing out synonyms, assessing overly specific collections, and filling in missing valence patterns are all improvements that can be guided by errors found by the GEDEM pipeline.
- *False synonyms* — Conversely, semtranses are sometimes too coarse-grained. For example, "lash", a synonym of "whip", maps to the collection `(CausingFn DamageOutcome)`. Many other verbs map to the same collection, including "bash", "break", and "squash", so *say* will falsely treat them as synonyms. This indicates that the ontology should be refined and the semtranses updated to capture more nuances. Paraphrase judgments are a good way to capture these kinds of errors.
- *Suspicion scores* — Manual analysis of suspicion scores shows the promise and limitations of our technique. Scores cluster at 0 and 1, which matches the small scale of the evaluation and the lack of overlap between clusters. We expect a smoother distribution for larger datasets that share more linguistic facts between sentences, something that can be achieved with targeted sampling and generation. The glut of facts with score 1 made the global ranked list difficult to browse, but analysis in the context of individual clusters proved useful. Typically, only one or two facts will separate the good sentences in a cluster from the bad ones, and having those sentences as context can help drill down on the errors. For example, a cluster containing "The dog has caught" and "The dog has catched" reveals an erroneous lexicon entry for "catched". Likewise, the appearance of "The horse is trampled" in the cluster in Figure 4 suggests CNLU and *say* fail to sufficiently distinguish between "has" and "is", a supposition backed up by other clusters in the dataset.

In general, our results provide preliminary evidence for GEDEM's usefulness as an error mining technique. The relative immaturity of *say* meant that many of the errors we found were in the early

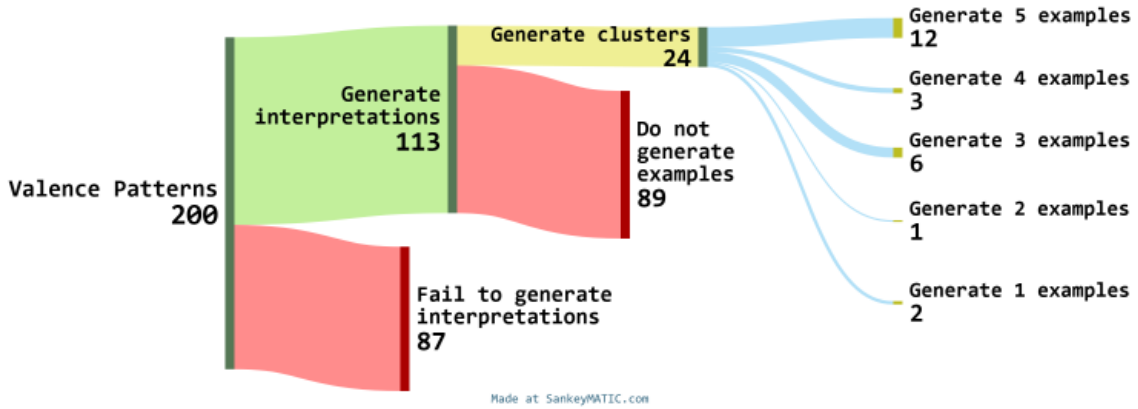

Figure 5. Results of paraphrase generation process

stages of the pipeline, but the errors it was able to uncover over even a small test will be useful for debugging CNLU and *say*. They also point to specific areas of improvement and tests that should be incorporated into our debugging toolkit (e.g., determining why synonyms like "whip" and "lash" are not treated as such by CNLU). As *say* becomes more reliable, we hope to turn GEDEM loose on larger datasets that better measure the frequency of errors in CNLU's linguistic knowledge and demonstrate its use at scale. Fixing the issues revealed by the current experiment, such as passive sentences and auxiliary verbs, will pave the way for uncovering new errors down the line.

6. Related Work

GEDEM is related to several lines of prior research. The sampling approach takes inspiration from Mittal and Moore (1996), who debugged ontological errors by sampling concrete examples for an expert to judge. GEDEM adapts this to linguistic knowledge by sampling semantics and generating examples from them. Whereas Mittal and Moore's approach automatically annotates its examples so that experts can check if the example matches its concept, GEDEM generates clusters of examples so that they can be compared to one another. In both cases, some additional context is needed for the user to tell if an example is "off": either a description of the concept the example belongs to, or a set of related examples that convey the intended meaning.

GEDEM also extends previous work on error mining by van Noord (2004), Sagot and de la Clergerie (2006), and de Kok and van Noord (2017). The basic approach is the same: run a hand-curated NL system over a set of examples, calculate an error signal, and use it to assign suspicion to errors that need expert attention to fix. However, where past work focused on syntactic errors using parsability as an automatic error signal, GEDEM uses clusters of generated sentences to expose errors in both semantic and syntactic knowledge, with a human in the loop to provide error judgments. The aim is to have an error signal that works for semantics but requires as little effort as possible to annotate.

Among error mining research, GEDEM is most similar to Egad (Goodman & Bond, 2009), an error mining system which parses, regenerates, and paraphrases sentences to find patterns of rules that need debugging. GEDEM also uses generation to expose errors in linguistic knowledge, but where Egad tests to see whether sentences *can* be parsed, regenerated, or paraphrased, GEDEM compares clusters of generated sentences to check their semantic content as well. Egad's use of round-trip generation fits nicely with INLD and GEDEM, and we plan to incorporate similar tests to help debug CNLU in the future.

Lastly, GEDEM is designed to complement Interactive Natural Language Debugging (Nakos & Forbus, 2024; Nakos, Kuthalam, & Forbus, 2022). Where INLD debugs individual sentences in depth, GEDEM uses a shallower approach to identify errors in bulk. In addition, INLD focuses on errors of restriction, which affect parsing more than generation, whereas GEDEM focuses on errors of permission, which affect generation more than parsing. Both frameworks aim to make hand-curated NL systems more maintainable.

7. Conclusion

In this paper, we introduced Generative Example-Driven Error Mining, a framework for finding errors in hand-curated NL systems. Traditional approaches to error mining have trouble dealing

with errors in semantic knowledge. GEDEM addresses this shortcoming by sampling semantic interpretations that can be produced by the parser, generating a cluster of example sentences for each interpretation, and asking the user to judge those examples. This lets the system assign suspicion to the linguistic knowledge associated with buggy generations. In aggregate, this allows GEDEM to highlight likely errors for system developers to fix.

We have presented an implemented version of GEDEM for CNLU, the language component of the Companion cognitive architecture, and introduced *say*, a new text generation system built on CNLU's linguistic knowledge. We have performed a preliminary demonstration of our approach by sampling a set of 200 valence patterns from CNLU and using them to generate example sentences to be judged. The GEDEM pipeline reveals errors at several stages, highlighting areas for improvement for CNLU and *say*. We also performed an initial experiment showing that LLMs may be able to provide error judgments in the future, making the GEDEM pipeline fully automatic.

Apart from improving *say* itself, there are several promising avenues for future work. The first, and most straightforward, is to expand the generative story beyond verb semtranses and valence patterns. While using them as semantic seeds is the most impactful way to clean up CNLU with the least complication, there are many parts of CNLU's linguistic knowledge that the current generative approach does not test. Adding in nouns, adjectives, quantifiers, modals, conjunctions, and other linguistic constructions will expand the reach of GEDEM and help it fulfill its purpose in cleaning up CNLU's knowledge and preparing it for use in text generation. Naturally, such extensions to GEDEM will go hand-in-hand with improvements to *say* to help bring it up to parity with CNLU. This forms a positive feedback loop, where targeted changes in GEDEM reveal errors that affect *say*, and fixing them expands the range of knowledge GEDEM can test.

Another avenue of future work is mining for errors of restriction. Because GEDEM reveals errors by generating sentences, it is limited to errors of permission, where the system's linguistic knowledge incorrectly permits faulty parse trees and semantic interpretations. Put another way, text generation can only reveal errors that lead to badly generated sentences, not ones that keep a correct sentence from being generated at all. Traditional error mining is sufficient to find errors of restriction in the grammar and lexicon by parsing text corpora, but semantic errors are harder to identify. Care must be taken to distinguish disambiguation errors, where the system produces the correct interpretation but does not select it, from errors in semantic knowledge, where the system cannot produce the correct interpretation at all. Currently, this remains an open challenge.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful suggestions. This research was supported by the Office of Naval Research.

References

Allen, J. (1994). *Natural language understanding* (2nd ed.). Redwood City, CA: Benjamin-Cummings Publishing Co., Inc.

de Kok, D., & van Noord, G. (2017). Mining for parsing failures. In *From Semantics to Dialectometry: Festschrift in Honor of John Nerbonne*. (p. 19.) College Publications.

Fillmore, C. J., Wooters, C., and Baker, C. F. (2001). Building a large lexical databank which provides deep semantics. *Proceedings of the 15th Pacific Asia Conference on Language, Information and Computation* (pp. 3-26). Hong Kong, China.

Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological Bulletin*, *76*, 378-382.

Forbus, K. D., & Hinrichs, T. (2017). Analogy and relational representation in the companion cognitive architecture. *AI Magazine*, *38*, 34-42.

Forbus, K., Usher, J., Lovett, A., Lockwood, K., & Wetzel, J. (2011). CogSketch: Sketch understanding for cognitive science research and for education. *Topics in Cognitive Science*, *3*(4), 648-666.

Goodman, M. W., & Bond, F. (2009). Using generation for grammar analysis and error detection. *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, (pp. 109-112).

Hovy, E., Marcus, M., Palmer, M., Ramshaw, L., & Weischedel, R. (2006). OntoNotes: the 90% solution. *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers* (pp. 57-60).

Lenat, D. B., & Guha, R. V. (1991). The evolution of CycL, the Cyc representation language. *ACM SIGART Bulletin*, *2*, 84-87.

Mittal, V. O., & Moore, J. D. (1996). Detecting knowledge base inconsistencies using automated generation of text and examples. *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conferences, AAAI 96, IAAI 96, 1*, (pp. 483-488). Portland.

Nakos, C., & Forbus, K. D. (2024). Interactively diagnosing errors in a semantic parser. *Proceedings of the Eleventh Annual Conference on Advances in Cognitive Systems*.

Nakos, C., & Forbus, K. D. (2025). Towards knowledge autonomy in the Companion cognitive architecture. *Cognitive Systems Research*, 101378. doi:10.1016/j.cogsys.2025.101378

Nakos, C., Kuthalam, M., & Forbus, K. D. (2022). A framework for interactive natural language debugging. *Proceedings of the Tenth Annual Conference on Advances in Cognitive Systems*.

Parsons, T. (1990). *Events in the semantics of English*. Cambridge, MA: MIT Press.

Sagot, B., & de la Clergerie, E. (2006). Error mining in parsing results. *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the ACL-ACL '06*.

Tintarev, N., & Masthoff, J. (2007). Effective explanations of recommendations: User-centered design. *Proceedings of the 2007 ACM Conference on Recommender Systems*, (pp. 153-156).

Tomai, E., & Forbus, K. D. (2009). EA NLU: Practical language understanding for cognitive modeling. *Proceedings of the 22nd International Florida Artificial Intelligence Research Society Conference*. Sanibel.

van Noord, G. (2004). Error mining for wide-coverage grammar engineering. *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, (pp. 446-453).

Wilson, J. R., Chen, K., Crouse, M., Nakos, C., Ribeiro, D. N., Rabkina, I., & Forbus, K. D. (2019). Analogical question answering in a multimodal information kiosk. *Proceedings of the Seventh Annual Conference on Advances in Cognitive Systems*. Cambridge.