Semi-Automated Protocol Disambiguation and Code Generation

Jane Yen
University of Southern California
yeny@usc.edu

Tamás Lévai
Budapest University of Technology and Economics
levait@tmit.bme.hu

Qinyuan Ye
University of Southern California
qinyuany@usc.edu

Xiang Ren
University of Southern California
xiangren@usc.edu

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Barath Raghavan
University of Southern California
barathra@usc.edu

ABSTRACT

For decades, Internet protocols have been specified using natural language. Given the ambiguity inherent in such text, it is not surprising that protocol implementations have long exhibited bugs. In this paper, we apply natural language processing (NLP) to effect semi-automated generation of protocol implementations from specification text. Our system, SAGE, can uncover ambiguous or under-specified sentences in specifications; once these are clarified by the author of the protocol specification, SAGE can generate protocol code automatically.

Using SAGE, we discover 5 instances of ambiguity and 6 instances of under-specification in the ICMP RFC; after fixing these, SAGE is able to automatically generate code that interoperates perfectly with Linux implementations. We show that SAGE generalizes to sections of BFD, IGMP, and NTP and identify additional conceptual components that SAGE needs to support to generalize to complete, complex protocols like BGP and TCP.

CCS CONCEPTS

• **Networks** → **Formal specifications**;

KEYWORDS

natural language, protocol specifications

ACM Reference Format:
Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. 2021. Semi-Automated Protocol Disambiguation and Code Generation. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3452296.3472910

1 INTRODUCTION

Four decades of Internet protocols have been specified in English and used to create, in Clark's words, rough consensus and running code [16]. In that time we have come to depend far more on network protocols than most imagined. To this day, engineers implement a protocol by reading and interpreting specifications as described in Request For Comments documents (RFCs). Their challenge is to navigate easy-to-misinterpret colloquial language while writing not only a bug-free implementation but also one that *interoperates* with code written by another person at a different time and place.

Software engineers find it difficult to interpret specifications in large part because natural language can be ambiguous. Unfortunately, such ambiguity is not rare; the errata alone for RFCs over the years highlight numerous ambiguities and the problems they have caused [17, 33, 68, 78]. Ambiguity has resulted in buggy implementations, security vulnerabilities, and has necessitated expensive and time-consuming software engineering processes, like interoperability bake-offs [32, 71].

To address this, one line of research has sought formal specification of programs and protocols (§8), which would enable verifying specification correctness and, potentially, enable automated code generation [13]. However, formal specifications are cumbersome and thus have not been adopted in practice; to date, protocols are specified in natural language.[1]

In this paper, we apply NLP to semi-automated generation of protocol implementations from RFCs. Our main challenge is to understand the *semantics* of a specification. This task, *semantic parsing*, has advanced in recent years with parsing tools such as CCG [5]. Such tools describe natural language with a *lexicon* and yield a semantic interpretation for each sentence. Because they are trained on generic prose, they cannot be expected to work out of the box for idiomatic network protocol specifications, which contain embedded syntactic cues (*e.g.,* structured descriptions of fields), incomplete sentences, and implicit context from neighboring text or other protocols. More importantly, the richness of natural language will likely always lead to ambiguity, so we do not expect fully-automated NLP-based systems (§2).

Contributions. In this paper, we describe SAGE, a *semi-automated* approach to protocol analysis and code generation from natural-language specifications. SAGE reads the natural-language protocol specification (*e.g.,* an RFC or Internet Draft) and marks sentences (a) for which it cannot generate unique semantic interpretations or (b) which fail on the protocol's *unit tests* (SAGE uses test-driven development). The former sentences are likely semantically ambiguous whereas the latter represent under-specified behaviors. In either case, the user (*e.g.,* the author of the specification) can then revise the sentences and re-run SAGE until the resulting RFC can cleanly be turned into code. SAGE can be used at various stages

SIGCOMM '21, August 23–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08.
https://doi.org/10.1145/3452296.3472910

[1] In recent years, attempts have been made to formalize other aspects of network operation, such as network configuration [7, 37] and control plane behavior [55], with varying degrees of success.

Name	Description
♦ **Packet Format**	Packet anatomy (i.e., field structure)
♦ **Field Descriptions**	Packet header field descriptions
♦ **Constraints**	Constraints on field values
♦ **Protocol Behaviors**	Reactions to external/internal events
System Architecture	Protocol implementation components
+ **State Management**	Session information and/or status
Comm. Patterns	Message sequences (e.g., handshakes)

Table 1: Protocol specification components. SAGE supports those marked with ♦ (fully) and + (partially).

in the standardization process (§2.3): while drafting, generating reference implementations, or revising a specification.

At the core of SAGE is an intermediate representation, called a *logical form*, of the semantics of a natural-language sentence. Intuitively, a logical form is a predicate expressing relationships between entities in the sentence. SAGE uses a logical form as a unifying abstraction underlying several tasks: (a) determining when a sentence may be fundamentally ambiguous, (b) identifying when to seek human input to expand its own vocabulary in order to parse the sentence, and (c) generating code.

SAGE is architected as a pipeline with three extensible stages, each of which makes unique contributions.

► The *parsing* stage (§3) generates logical forms for each input sentence. To do this, SAGE extends a pre-existing semantic parser ([75]) with domain-specific constructs necessary to correctly parse IETF standards. These constructs include networking-specific vocabulary and domain-specific semantics (*e.g.,* the use of the word "is" to specify assignment). SAGE includes tools that we developed to parse structural context (*e.g.,* indentation to specify field descriptions) and non-textual elements (*e.g.,* ASCII art for packet header representations).

► Ideally, the parser should be able to reduce each sentence to a single logical form. In practice, RFCs contain idiomatic usage that confounds natural language parsers, such as incomplete sentences to describe protocol header fields and specific uses of verbs like *is* and prepositions like *of*. For these sentences, the parser may emit multiple logical forms. SAGE's *disambiguation* stage contains multiple checks that *filter* out logical forms that incorrectly interpret this idiomatic usage. We have developed these filters in the course of using SAGE to parse RFCs. Even so, at the end of this stage, a sentence may not result in a single logical form either (a) because the parser's vocabulary or the disambiguation stage's filters are incomplete, or (b) the sentence may be fundamentally ambiguous. SAGE prompts the user to extend the vocabulary or add a filter (for (a)) or rewrite the sentence (for (b)). As users repeatedly extend ("train") SAGE's vocabulary and filters by parsing RFCs, we expect the level of human involvement to drop significantly (§2.3).

► Once each sentence has been reduced to a single logical form, SAGE's *code generator* converts semantic representations to executable code (§5). To do this, the code generator uses contextual information that it has gleaned from the RFC's document structure, as well as static context predefined in SAGE about lower-layer protocols and the underlying OS. Unit testing on generated code can uncover incompleteness in specifications.

SAGE discovered (§6) 5 sentences in the ICMP RFC [63] (of which 3 are unique, the others being variants) that had multiple semantic interpretations even after disambiguation. It also discovered 6 sentences that failed unit tests (all variants of a single sentence). After we rewrote these sentences, SAGE was able to automatically generate code for ICMP that interoperated perfectly with `ping` and `traceroute`. In contrast, graduate students asked to implement ICMP in a networking course made numerous errors (§2). Moreover, SAGE was able to parse sections of BFD [35], IGMP [19], and NTP [54] (but does not yet fully support these protocols), with few additions to the lexicon. It generated packets for the timeout procedure containing both NTP and UDP headers. It also parsed state management text for BFD to determine system actions and update state variables for reception of control packets. Finally, SAGE's disambiguation is often very effective, reducing, in some cases, 56 logical forms (an intermediate representation) to 1. We have open-sourced our SAGE implementation [69].

Toward greater generality. SAGE is a significant first step toward automated processing of natural-language protocol specifications, but much work remains. Protocol specifications contain many components; Table 1 indicates which ones SAGE supports well (in green), which it supports partially (in olive), and which it does not support. Some protocols contain complex state machine descriptions (*e.g.,* TCP) or describe how to process and update state (*e.g.,* BGP); SAGE can parse state management in a simpler protocol like BFD. Other protocols describe software architectures (*e.g.,* OSPF, RTP) and communication patterns (*e.g.,* BGP); SAGE must be extended to parse these descriptions. In §7, we break down the prevalence of protocol components by RFC to contextualize our contributions, and identify future SAGE extensions. Such extensions will put SAGE within reach of parsing large parts of TCP and BGP RFCs.

Broader implications. We note three broader takeaways from our work on SAGE. First, we wish to highlight the consequences of ambiguity in specifications and how they can manifest in code. Second, with a proper analysis and disambiguation tool (i.e. CCG lexicons and disambiguation checks), SAGE can highlight ambiguities for RFC authors, editors, protocol developers, etc. Third, SAGE shows the feasibility of generating specification code from natural language descriptions, and we hope SAGE can inspire future work to overcome the code generation challenges of diverse natural-language contexts.

2 BACKGROUND AND OVERVIEW

Specification ambiguities can lead to bugs and non-interoperability, which we quantify using implementations of ICMP [63] by students in a graduate networking course.

2.1 Discussion of ICMP Implementations

ICMP, defined in RFC 792 in 1981 and used by core tools like `ping` and `traceroute`, is a simple protocol whose specification should be easy to interpret. To test this assertion, we examined implementations of ICMP by 39 students in a graduate networking class. Given the ICMP RFC and related RFCs, students built ICMP message handling for a router.[2]

[2]Ethics note: the code artifacts we examined were pre-existing, with no personal or identifying information; the code was not generated for this analysis.

Error Type	Frequency
IP header related	57%
ICMP header related	57%
Network byte order and host byte order conversion	29%
Incorrect ICMP payload content	43%
Incorrect echo reply packet length	29%
Incorrect checksum or dropped by kernel	36%

Table 2: Error types of failed cases and their frequency in 14 faulty student ICMP implementations.

Index	ICMP checksum range interpretations
1	Size of a specific type of ICMP header.
2	Size of a partial ICMP header.
3	Size of the ICMP header and payload.
4	Size of the IP header.
5	Size of the ICMP header and payload, and any IP options.
6	Incremental update of the checksum field using whichever checksum range the sender packet chose.
7	Magic constants (e.g., 2 or 8 or 36).

Table 3: Students' ICMP checksum range interpretations.

To test whether students implemented echo reply correctly, we used the Linux `ping` tool to send an echo message to their router (we tested their code using Mininet [44]). Across the 39 implementations, the Linux implementation correctly parsed the echo reply only for 24 of them (61.5%). One failed to compile and the remaining 14 exhibited 6 categories (not mutually exclusive) of implementation errors (Table 2): mistakes in IP or ICMP header operations; byte order conversion errors; incorrectly-generated ICMP payload in the echo reply message; incorrect length for the payload; and wrongly-computed ICMP checksum. Each error category occurred in at least 4 of the 14 erroneous implementations.

To understand the incorrect checksum better, consider the specification of the ICMP checksum in this sentence: *The checksum is the 16-bit one's complement of the one's complement sum of the ICMP message starting with the ICMP Type.* This sentence does not specify where the checksum should *end*, resulting in a potential ambiguity for the echo reply; a developer could checksum some or all of the header, or both the header and the payload. In fact, students came up with *seven* different interpretations (Table 3) including checksumming only the IP header, checksumming the ICMP header together with a few fixed extra bytes, and so on.

2.2 Approach

Dealing with Ambiguity. Students in an early graduate course might be expected to make mistakes in implementing protocols from specifications, but we were surprised at the prevalence of errors (Table 2) and the range of interpretations of parts of the specification (Table 3) in student code. We do not mean to suggest that seasoned protocol developers would make similar mistakes. However, this exercise highlights why RFC authors and the IETF community have long relied on manual methods to avoid or eliminate non-interoperabilities: careful review of standards drafts by participants, development of reference implementations, and interoperability *bake-offs* [32, 71] at which vendors and developers test their implementations against each other to discover issues that often arise from incomplete or ambiguous specifications.

Why are there ambiguities in RFCs? RFCs are ambiguous because (a) natural language is expressive and admits multiple ways to express a single idea; (b) standards authors are technical domain experts who may not always recognize the nuances of natural language; and (c) context matters in textual descriptions, and RFCs may omit context.

Can reference implementations alone eliminate ambiguity? Reference implementations are useful but insufficient. For a reference protocol document to become a standard, a reference implementation is indeed often written, and this has been the case for many years. A reference implementation is often written by participants in the standardization process, who may or may not realize that there exist subtle ambiguities in the text. Meanwhile, vendors write code directly to the specification (often to ensure that the resulting code has no intellectual property encumbrances), sometimes many years after the specification was standardized. This results in subtle incompatibilities in implementations of widely deployed protocols [59].

Approach: Semi-automated Semantic Parsing of RFCs. Unlike general English text, network protocol specifications have exploitable structure. The networking community uses a restricted set of words and operations (*i.e., domain-specific* terminology) to describe network behaviors. Moreover, RFCs conform to a uniform style [22] (especially recent RFCs) and all standards-track RFCs are carefully edited for clarity and style adherence [67].

Motivated by this observation, we leverage recent advances in the NLP area of *semantic parsing*. Natural language can have lexical [36, 66] (*e.g.,* the word *bat* can have many meanings), structural (*e.g.,* the sentence *Alice saw Bob with binoculars*) and semantic (*e.g.,* in the sentence *I saw her duck*) ambiguity. Semantic parsing tools can help identify these ambiguities. However, for the foreseeable future we do not expect NLP to be able to parse RFCs without some human input. Thus, SAGE is *semi-automated* and uses NLP tools, along with unit tests, to help a human-in-the-loop discover and correct ambiguities after which the specification is amenable to automated code generation.

2.3 SAGE Overview

Figure 1 shows the three stages of SAGE. The *parsing* stage uses a semantic parser [5] to generate intermediate representations, called *logical forms* (LFs), of sentences. Because parsing is not perfect, it can output multiple LFs for a sentence. Each LF corresponds to one semantic interpretation of the sentence, so multiple LFs represent ambiguity. The *disambiguation* stage aims to automatically eliminate such ambiguities. If, after this, ambiguities remain, SAGE asks a human to resolve them. The *code generator* compiles LFs into executable code, a process that may also uncover ambiguity.

SAGE Workflow. To clarify how SAGE works, and when (and for what reason) human involvement is necessary, we briefly describe the workflow that a SAGE user (*e.g.,* a specification author) would follow (Figure 2). First, the user extracts actionable sections of a specification and feeds these to the semantic parsing stage. RFCs contain significant explanatory, non-actionable, material (*e.g.,* the introduction) that may not be relevant to the analysis; SAGE currently requires a human to identify these, but can potentially identify such sections automatically, which we have left to future work. The parsing stage analyzes each sentence in the input. The output of this stage is a set of logical forms representing semantic

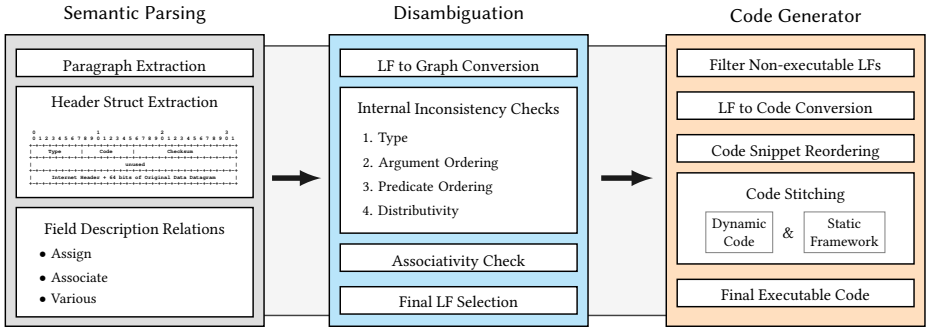


Figure 1: SAGE components.

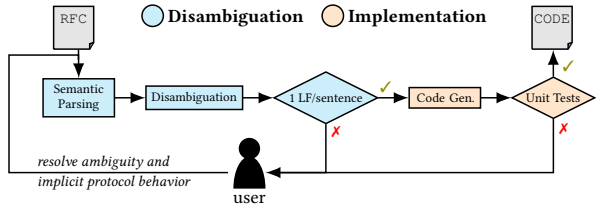


Figure 2: SAGE workflow in processing RFC 792.

interpretations of the sentence (§3). The disambiguation stage (§4) winnows these logical forms based on built-in checks that capture domain-specific usage in protocol specifications.

If this step does not result in a single LF, there are two possibilities: (a) either the sentence is fundamentally ambiguous, or (b) the sentence contains terms not present in SAGE's lexicon or domain specific usage not present in SAGE's built-in checks. At this point, SAGE presents the sentence to the user, who can, for case (a), rewrite the sentence to resolve the ambiguity, or, for case (b), extend SAGE's lexicon or add to its built-in checks. This is akin to systems like spell and grammar checkers, which present users with potential errors, and permit users to add entries to local dictionaries as part of a correction step. Adding new lexical entries is, of course, more difficult than adding entries to a dictionary. Our SAGE implementation contains a simple user interface enhancement to suggest additions in order to reduce the cognitive load on the user. Better user interfaces can further reduce cognitive load, but will require significant user studies so we leave these to future work.

Over time, as SAGE is used to analyze RFCs, we expect this manual effort to decline significantly. Our intuition for this comes from Zipf's law, first defined in quantitative linguistics, which shows that the frequency of word usage is heavy-tailed: some are very common while others are rare. Over time, the lexical entries and checks added to SAGE may cover most of the text in a new specification, and users need only add the occasional lexical entry or domain-specific check. Our evaluations (§6) corroborate this intuition.

Once each sentence has been reduced to a single LF, the code generator stage (§5) generates protocol code and runs unit tests on them. These unit tests are to be written by the spec author; SAGE employs test-driven development (§6.5). If a unit test fails, it is likely that protocol behavior is under-specified. At this point as well SAGE notifies the user (*e.g.,* the specification author), who can rewrite the relevant sentence(s) and re-invoke the entire pipeline.

How and when to use SAGE. A standards document begins its life as an Internet Draft discussed at several IETF meetings. At this stage, specification authors can use SAGE to identify fundamentally ambiguous sentences. Before the protocol is standardized, participants in the standardization process develop a reference implementation. During this stage, developers of the reference implementation can test SAGE's auto-generated code against their implementation to identify under-specified behavior (§6.5). Finally, when a vendor decides to implement the protocol on their platform, they can use SAGE's generated code as a starting point for their implementation.

SAGE can also help to revise specifications in two ways. Its disambiguation stage can eliminate ambiguity introduced during the revision. Moreover, it can generate code for two different versions of a specification, and with the help of analysis tools (*e.g.,* static analysis, control flow analysis), a future version of SAGE could help protocol implementers to develop backward compatibility mechanisms between the two versions.

3 SEMANTIC PARSING

Semantic parsing is the task of extracting meaning from a document. Tools for semantic parsing formally specify natural language grammars and extract parse trees from text. More recently, deep-learning based approaches have proved effective in semantic parsing [21, 41, 88] and certain types of automatic code generation [47, 64, 86]. However, such methods do not directly apply to our task. First, deep learning typically requires training in a "black-box". Since we aim to identify ambiguity in specifications, we aim to interpret intermediate steps in the parsing process and maintain all valid parsings. Second, such methods require large-scale annotated datasets; collecting high-quality data that maps network protocol specifications to expert-annotated logical forms (for supervised learning) is impractical.

For these reasons, we use the Combinatory Categorial Grammar (CCG [5]) formalism that enables (a) coupling syntax and semantics in the parsing process and (b) is well suited to handling domain-specific terminology by defining a small hand-crafted lexicon that encapsulates domain knowledge. CCG has been used to parse natural language explanations into labeling rules in several contexts [74, 82].

CCG background. A CCG takes as input a description of the language syntax and semantics. It describes the syntax of words and phrases using *primitive categories* such as noun (N), noun phrase (NP), or sentence (S), and *complex categories* comprised of primitive categories, such as S\NP (to express that it can combine a noun

phrase on the left and form a sentence). It describes semantics with lambda expressions such as $\lambda x.\lambda y.@\text{Is}(y, x)$ and $\lambda x.@\text{Compute}(x)$.

CCG employs a *lexicon*, which users can extend to capture domain-specific knowledge. For example, we added the following lexical entries to the lexicon to represent constructs found in networking standards documents:

(1) checksum → NP: "checksum"
(2) is → {(S\NP)/NP: $\lambda x.\lambda y.@\text{Is}(y, x)$}
(3) zero → {NP: @Num(0)}

This expresses the fact (a) "checksum" is a special word in networking, (b) "is" can be assignment, and (c) zero can be a number. CCG can use this lexicon to generate a *logical form* (LF) that completely captures the semantics of a phrase such as "checksum is zero": {S: @Is("checksum", @Num(0))}. Our code generator (§5) produces code from these.

Challenges. SAGE must surmount three challenges before using CCG: (a) specify *domain-specific* syntax, (b) specify *domain-specific* semantics, (c) extract *structural* and *non-textual* elements in standards documents (described below). Next we describe how we address these challenges.

Specifying domain-specific syntax. Lexical entry (1) above specifies that *checksum* is a keyword in the vocabulary. Rather than having a person specify such syntactic lexical entries, SAGE creates a *term dictionary* of domain-specific nouns and noun-phrases using the index of a standard networking textbook. This reduces human effort. Before we run the semantic parser, we also need to identify nouns and noun-phrases that occur generally in English, for which we use an NLP tool called SpaCy [29].

Specifying domain-specific semantics. NLTK's CCG [49] has a built-in lexicon that captures the semantics of written English. Even so, we have found it important to add domain-specific lexical entries. For example, the lexical entry (2) above shows that the verb *is* can represent the assignment of a value to a protocol field. In SAGE, we manually generate these domain-specific entries, with the intent that these semantics will generalize to many RFCs (see also §6). Beyond capturing domain-specific uses of words (like *is*), domain-specific semantics capture idiomatic usage common to RFCs. For example, RFCs have field descriptions (like version numbers, packet types) that are often followed by a single sentence that has the (fixed) value of the field. For a CCG to parse this, it must know that the value should be assigned to the field. Similarly, RFCs sometimes represent descriptions for different code values of a type field using an idiom of the form "0 = Echo Reply". §6 quantifies the work involved in generating the domain-specific lexicon.

Extracting structural and non-textual elements. Finally, RFCs contain stylized elements, for which we wrote pre-processors. RFCs use descriptive lists (*e.g.,* field names and their values) and indentation to note content hierarchy. Our pre-processor extracts these relationships to aid in disambiguation (§4) and code generation (§5). RFCs also represent header fields (and field widths) with ASCII art; we extract field names and widths and generate data structures (specifically, structs in C) to represent headers to enable automated code generation (§5). Some RFCs [54] also contain pseudo-code, which we represent as logical forms to facilitate code generation.

Running a CCG. After pre-processing, we run a CCG on each sentence of an RFC. Ideally, a CCG should output exactly one logical form for a sentence. In practice, it outputs ***zero or more*** logical forms, some of which arise from CCG limitations, and some from ambiguities inherent in the sentence.

4 DISAMBIGUATION

Next we describe how SAGE leverages domain knowledge to automatically resolve some ambiguities, where semantic parsing resulted in either 0 or more than 1 logical forms.

4.1 Why Ambiguities Arise

To show how we automatically resolve ambiguities, we take examples from the ICMP RFC [63] for which our semantic parser returned either 0 or more than 1 logical forms.

Zero logical forms. Several sentences in the ICMP RFC resulted in zero logical forms after semantic parsing, all of which were grammatically incomplete, lacking a subject:

A *The source network and address from the original datagram's data*
B *The internet header plus the first 64 bits of the original datagram's data*
C *If code = 0, identifies the octet where an error was detected*
D *Address of the gateway to which traffic for the network specified in the internet destination network field of the original datagram's data should be sent*

Such sentences are common in protocol header field descriptions. The last sentence is difficult even for a human to parse.

More than 1 logical form. Several sentences resulted in more than one logical form after semantic parsing. The following two sentences are *grammatically incorrect*:

E *If code = 0, an identifier to aid in matching timestamp and replies, may be zero*
F *If code = 0, a sequence number to aid in matching timestamp and replies, may be zero*

The following example needs *additional context*, and contains *imprecise language*:

G *To form a information reply message, the source and destination addresses are simply reversed, the type code changed to 16, and the checksum recomputed*

A machine parser does not realize that source and destination addresses refer to fields in the IP header. Similarly, it is unclear from this sentence whether the checksum refers to the IP checksum or the ICMP checksum. Moreover, the term *type code* is confusing, even to a (lay) human reader, since the ICMP header contains both a *type* field and a *code* field.

Finally, this sentence, discussed earlier (§2.1), is under-specified, since it does not describe which byte the checksum computation should *end* at:

H *The checksum is the 16-bit ones's complement of the one's complement sum of the ICMP message starting with the ICMP Type*

While sentences ***G*** and ***H*** are grammatically correct and should have resulted in a single logical form, the CCG parser considers them ambiguous as we explain next.

Causes of ambiguities: zero logical forms. Examples ***A*** through ***C*** are missing a subject. In the common case when these sentences describe a header field, that header field is usually the subject of the sentence. This information is available to SAGE when it extracts structural information from the RFC (§3). When a sentence that is part of a field description has zero logical forms, SAGE can re-parse that sentence by supplying the header. This approach does not work for ***D***; this is an incomplete sentence, but CCG is unable to parse it even with the supplied header context. Ultimately, we had to re-write that sentence to successfully parse it.

Causes of ambiguities: more than one logical form. Multiple logical forms arise from more fundamental limitations in machine parsing. Consider Figure 3, which shows multiple logical forms arising for a single sentence. Each logical form consists of *nested predicates* (similar to a statement in a functional language), where each predicate has one or more arguments. A predicate represents a logical relationship (@AND), an assignment (@Is), a conditional (@IF), or an action (@ACTION) whose first argument is the name of a function, and subsequent arguments are function parameters. Finally, Figure 3 illustrates that a logical form can be naturally represented as a tree, where the internal nodes are predicates and leaves are (scalar) arguments to predicates.

Inconsistent argument types. In some logical forms, their arguments are incorrectly typed, so they are obviously wrong. For example, LF1 in Figure 3, the second argument of the `compute` action must be the name of a function, not a numeric constant. CCG's lexical rules don't support type systems, so cannot eliminate badly-typed logical forms.

Order-sensitive predicate arguments. The parser generates multiple logical forms for the sentence ***E***. Among these, in one logical form, `code` is assigned zero, but in the others, the `code` is tested for zero. Sentence ***E*** has the form "If A, (then) B", and CCG generates two different logical forms: @IF(A,B) and @IF(B,A). This is not a mistake humans would make, since the condition and action are clear from the sentence. However, CCG's flexibility and expressive power may cause over-generation of semantic interpretations in this circumstance. This unintended behavior is well-known [28, 83].

Predicate order-sensitivity. Consider a sentence of the form "A of B is C". In this sentence, CCG generates two distinct logical forms. In one, the @OF predicate is at the root of the tree, in the other @Is is at the root of the tree. The first corresponds to the grouping "(A of B) is C" and the second to the grouping "A of (B is C)". For sentences of this form, the latter is incorrect, but CCG unable to generate disambiguate between the two.

Predicate distributivity. Consider a sentence of the form "A and B is C". This sentence exemplifies a grammatical structure called *coordination* [75][3]. For such a sentence, CCG will generate two logical forms, corresponding to: "(A and B) is C" and "(A is C) and (B is C)" (in the latter form, "C" distributes over "A" and "B"). In general, both forms are equally correct. However, CCG sometimes chooses to distribute predicates when it should not. This occurs because CCG is unable to distinguish between two uses of the *comma*: one as a conjunction, and the other to separate a dependent clause from an independent clause. In sentences with a comma, CCG generates logical forms for both interpretations. RFCs contain some sentences of the form "A, B is C"[4]. When CCG interprets the comma to mean a conjunction, it generates a logical form corresponding to "A is C and B is C", which, for this sentence, is clearly incorrect.

Predicate associativity. Consider sentence ***H***, which has the form "A of B of C", where each of A, B, and C are predicates (*e.g.,* A is the predicate @ACTION("16-bit-ones-complement"). In this example, the CCG parser generates two semantic interpretations corresponding to two different groupings of operations (one that groups A and B, the other that groups B and C: Figure 4). In this case, the @OF predicate is associative, so the two logical forms are equivalent, but the parser does not know this.

4.2 Winnowing Ambiguous Logical Forms

We define the following checks to address each of the above types of ambiguities (§4.1), which SAGE applies to sentences with multiple logical forms, winnowing them down (often) to one logical form (§6). These checks apply broadly because of the restricted way in which specifications use natural language. While we derived these by analyzing ICMP, we show that these checks also help disambiguate text in other RFCs. At the end of this process, if a sentence is still left with multiple logical forms, it is fundamentally ambiguous, so SAGE prompts the user to re-write it.

Type. For each predicate, SAGE defines one or more type checks: action predicates have function name arguments, assignments cannot have constants on the left hand side, conditionals must be well-formed, and so on.

Argument ordering. For each predicate for which the order of arguments is important, SAGE defines checks that remove logical forms that violate the order.

Predicate ordering. For each pair of predicates where one predicate cannot be nested within another, SAGE defines checks that remove order-violating logical forms.

Distributivity. To avoid semantic errors due to comma ambiguity, SAGE always selects the non-distributive logical form version (in our example, "(A and B) is C").

Associativity. If predicates are associative, their logical form trees (Figure 4) will be *isomorphic*. SAGE detects associativity using a standard graph isomorphism algorithm.

5 CODE GENERATION

Next we discuss how we convert the intermediate representation of disambiguated logical forms to code.

5.1 Challenges

We faced two main challenges in code generation: (a) representing implicit knowledge about dependencies between two protocols or

[3]For example: *Alice sees and Bob says he likes Ice Cream.*

[4]If a higher-level protocol uses port numbers, they are assumed to be in the first 64 data bits of the original datagram's data.

Sentence	*For computing the checksum, the checksum field should be zero*
LF 1	@AdvBefore(@Action('compute','0'),@Is(@And('checksum_field','checksum'),'0'))
LF 2	**@AdvBefore(@Action('compute','checksum'),@Is('checksum_field','0'))**
LF 3	@AdvBefore('0',@Is(@Action('compute',@And('checksum_field','checksum')),'0'))
LF 4	@AdvBefore('0',@Is(@And('checksum_field',@Action('compute','checksum')),'0'))

LF 2:

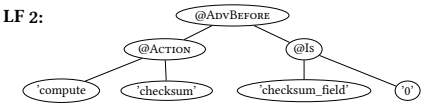


Figure 3: Example of multiple LFs from CCG parsing of "For computing the checksum, the checksum should be zero".

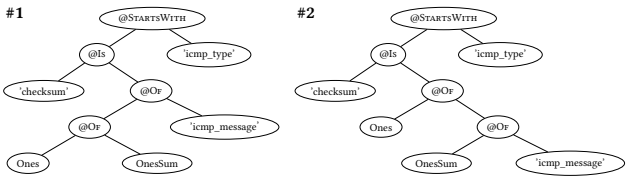


Figure 4: LF Graphs of sentence *H*.

a protocol and the OS and (b) converting a functional logical form into imperative code.

Encoding protocol and environment dependencies. Networked systems rely upon protocol stacks, where protocols higher in the stack use protocols below them. For example, ICMP specifies what operations to perform on IP header fields (*e.g.,* sentence ***G*** in §4), and does not specify but assumes an implementation of one's complement. Similarly, standards descriptions do not explicitly specify what abstract functionality they require of the underlying operating system (*e.g.,* the ability to read interface addresses).

To address this challenge, SAGE requires a pre-defined *static framework* that provides such functionality along with an API to access and manipulate headers of other protocols, and to interface with the OS. SAGE's generated code (discussed below) uses the static framework. The framework may either contain a complete implementation of the protocols it abstracts, or, more likely, invoke existing implementations of these protocols and services provided by the OS.

Logical Forms as an Intermediate Representation. The parser generates an LF to represent a sentence. For code generation, these sentences (or fragments thereof) fall into two categories: actionable and non-actionable sentences. Actionable sentences result in executable code: they describe value assignments to fields, operations on headers, and computations (*e.g.,* checksum). Non-actionable sentences do not specify executable code, but specify a future intent such as *"The checksum may be replaced in the future"* or behavior intended for other protocols such as *"If a higher level protocol uses port numbers, port numbers are assumed to be in the first 64 data bits of the original datagram's data"*. Humans may intervene to identify non-actionable sentences; SAGE tags their logical forms with a special predicate @AdvComment.

The second challenge is that parsers generate logical forms for individual sentences, but the ordering of code generated from these logical forms is not usually explicitly specified. Often the order in which sentences occur matches the order in which to generate code for those sentences. For example, an RFC specifies how to set field values, and it is safe to generate code for these fields in the order in which they appear. There are, however, exceptions to this. Consider the sentence in Figure 3, which specifies that, when computing the checksum, the checksum field must be zero. This sentence occurs in the RFC *after* the sentence that describes how to compute

LF	@Is('type', '3')
context	{"protocol": "ICMP", "message": "Destination Unreachable Message", "field": "type", "role": ""}
code	`hdr->type = 3;`

Table 4: Logical form with context and resulting code.

checksum, but its executable code must occur *before*. To address this, SAGE contains a lexical entry that identifies, and appropriately tags (using a special predicate @AdvBefore), sentences that describe such *advice* (as used in functional and aspect-oriented languages).[5]

5.2 Logical Forms to Code

Pre-processing and contextual information. The process of converting logical forms to code is multi-stage, as shown in the right block of Figure 1. Code generation begins with pre-processing actions. First, SAGE filters out logical forms with the @AdvComment predicate. Then, it prepares logical forms for code conversion by adding contextual information. A logical form does not, by itself, have sufficient information to auto-generate code. For example, from a logical form that says 'Set (message) type to 3' (@Is(type, 3)) it is not clear what "type" means and must be inferred from the context in which that sentence occurs. In RFCs, this context is usually implicit from the document structure (the section, paragraph heading, or indentation of text). SAGE auto-generates a *context dictionary* for each logical form (or sentence) to aid code generation (Table 4).

In addition to this dynamic context, SAGE also has a *pre-defined static context dictionary* that encapsulates information in the static context. This contains field names used in lower-level protocols (*e.g.,* the table maps terms source and destination addresses to corresponding fields in the IP header, or the term "one's complement sum" to a function that implements that term). During code generation, SAGE first searches the dynamic context, then the static context.

Code generation. After preprocessing, SAGE generates code for a logical form using a post-order traversal of the single logical form obtained after disambiguation. For each predicate, SAGE uses the context to convert the predicate to a code snippet using both a dictionary of predicate-code snippet mappings and contextual information; concatenating these code snippets results in executable code for the logical form. For corner-cases, SAGE applies user-defined conversions to fine-tune the resulting code.

SAGE then concatenates code snippets for all the logical forms in a message into a packet handling function[6]. In general, for a given message, it is important to distinguish between code executed at the sender versus at the receiver, and to generate two functions, one at the sender and one at the receiver. Whether a logical form

[5] Advice covers statements associated with a function that must be executed before, after, or instead of that function. Here, the checksum must be set to zero *before* computing the checksum.

[6] SAGE generated code examples are available at [69].

applies to the sender or the receiver is also encoded in the context dictionary (Table 4). Also, SAGE uses the context to generate unique names for the function, based on the protocol, the message type, and the role, all of which it obtains from the context dictionaries.

Finally, SAGE processes advice at this stage to decide on the order of the generated executable code. In its current implementation, it only supports @AdvBefore, which inserts code before the invocation of a function.

These functions are inserted into a static framework at code stitching (Figure 1). This framework provides required networking functions such as I/O handling involving socket management or, for testing purposes, PCAP read/write and helper functions (*e.g.,* parity checks, checksum calculation).

Adapting the code generator to new protocol packet handling functions might require some human effort in updating the conversion tables. Additionally, new predicates need to be added to the predicate-code snippet mapping when they are first introduced. We found these steps require no deep protocol knowledge since most of the rules are general. Significant engineering effort is only required for implementing helper functions for the static framework, which we expect will be rare after a larger library of these is developed.

Iterative discovery of non-actionable sentences. Non-actionable sentences are those for which SAGE should not generate code. Rather than assume that a human annotates each RFC with such sentences before SAGE can execute, SAGE provides support for *iterative discovery* of such sentences, using the observation that *a non-actionable sentence will usually result in a failure during code generation.* So, to discover such sentences, a user runs the RFC through SAGE repeatedly. When it fails to generate code for a sentence, it alerts the user to confirm whether this was a non-actionable sentence or not, and annotates the RFC accordingly. During subsequent passes, it tags the sentence's logical forms with @AdvComment, which the code generator ignores.

In ICMP, for example, there are 35 such sentences. Among RFCs we evaluated, SAGE can automatically tag such code generation failures as @AdvComment without human intervention (*i.e.,* there were no cases of an actionable sentence that failed code generation once we defined the context).

6 EVALUATION

Next we quantify SAGE's ability to find specification ambiguities, its generality across RFCs, and the importance of disambiguation and of our parsing and code generation extensions.

6.1 Methodology

Implementation. SAGE includes a networking dictionary, new CCG-parsable lexicon entries, a set of inconsistency checks, and LF-to-code predicate handler functions. We used the index of [42] to create a dictionary of about 400 terms. SAGE adds 71 lexical entries to an NLTK-based CCG parser [49]. [7] Overall, SAGE consists of 7,128 lines of code. In addition, the static framework is 1478 lines of code; this framework is reused across all protocols.

To winnow ambiguous logical forms for ICMP (§4.2), we defined 32 type checks, 7 argument ordering checks, 4 predicate ordering checks, and 1 distributivity check. Argument ordering and predicate ordering checks maintain a blocklist. Type checks use an allowlist and are thus the most prevalent. The distributivity check has a single implicit rule. For code generation, we defined 25 predicate handler functions to convert LFs to code snippets. As we analyzed additional protocols (IGMP, NTP and BFD), we manually added more lexical entries and type checks, using the workflow described in §2.3; we quantify the overhead of these in §6.3 and §6.4. Across all of these protocols, SAGE auto-generated 554 lines of protocol code after disambiguation.

Test Scenarios. First we examine the ICMP RFC, which defines 8 ICMP message types.[8] Like the student assignments we analyzed earlier, we generated code for each ICMP message type. To test this for each message, as with the student projects, the client sends test messages to the router which then responds with the appropriate ICMP message. For each scenario, we captured both sender and receiver packets and verified correctness with `tcpdump`. We include details of each scenario in the Appendix. To demonstrate the generality of SAGE, we also evaluated IGMP, NTP, and BFD.

6.2 End-to-end Evaluation

Next we verify that ICMP code generated by SAGE produces packets that interoperate correctly with Linux tools.

Packet capture based verification. In the first experiment, we examined the packet emitted by a SAGE-generated ICMP implementation with `tcpdump` [76], to verify that `tcpdump` can read packet contents correctly without warnings or errors. Specifically, for each message type, for both sender and receiver side, we use the static framework in SAGE-generated code to generate and store the packet in a pcap file and verify it using `tcpdump`. `tcpdump` output lists packet types (*e.g.,* an IP packet with a time-exceeded ICMP message) and will warn if a packet of truncated or corrupted packets. In all of our experiments we found that ***SAGE generated code produces correct packets with no warnings or errors***.

Interoperation with existing tools. Here we test whether a SAGE-generated ICMP implementation interoperates with tools like `ping` and `traceroute`. To do so, we integrated our static framework code and the SAGE-generated code into a Mininet-based framework used for the course described in §2. With this framework, we verified, with four Linux commands (testing echo, destination unreachable, time exceeded, and traceroute behavior), that a SAGE-generated receiver or router correctly processes echo request packets sent by `ping` and TTL-limited data packets or packets to non-existent destinations sent by `traceroute`, and its responses are correctly interpreted by those programs. For all these commands, the ***generated code interoperates correctly with these tools***. We also conducted interoperability experiments on real machines. To do so, we extended our static framework to send and receive ICMP packets on raw sockets. The result was identical to our Mininet experiments.

[7] NLTK is a popular general-purpose NLP toolkit: over 100k+ GitHub repositories depend on it [57]. We are aware of limitations of NLTK's CCG parser; other tools such as SPF [4] may address these limitations. We leave the comparison of the two toolkits and possible migration to SPF to future work.

[8] ICMP message types include destination unreachable, time exceeded, parameter problem, source quench, redirect, echo/echo reply, timestamp/timestamp reply, and information request/reply.

6.3 Exploring Generality: IGMP and NTP

To understand the degree to which SAGE generalizes to other protocols, we ran it on two other protocols: parts of IGMP v1 as specified in RFC 1112 [19] and NTP [54]. These RFCs contain conceptual elements such as architecture description and behavior not specific to network protocols (*e.g.,* NTP stratums). These are currently not supported by SAGE. In §7, we discuss what it will take to extend SAGE to completely parse these RFCs and generalize it to a larger class of protocols.

IGMP. In RFC 1112 [19], we parsed the packet header description in Appendix I of the RFC. To do this, we added to SAGE 8 lexical entries (beyond the 71 we had added for ICMP entries), 4 predicate function handlers (from 21 for ICMP), and 1 predicate ordering check (from 7 for ICMP). For IGMP, SAGE generates the sending of host membership and query message. We also verified interoperability of the generated code. In our test, our generated code sends a host membership query to a commodity switch. We verified, using packet captures, that the switch's response is correct, indicating that it interoperates with the sender code.

NTP. For NTP [54], we parsed Appendices A and B: these describe, respectively, how to encapsulate NTP messages in UDP, and the NTP packet header format and field descriptions. To parse these, we added only 5 additional lexical entries and 1 predicate ordering check beyond what we already had for IGMP and ICMP.

6.4 Exploring Generality: BFD

Thus far, we have discussed how SAGE supports headers, field descriptions, constraints, and basic behaviors. We now explore applying SAGE to BFD [35], a recent protocol whose specification contains sentences that describe how to initiate/update state variables. We have used SAGE to parse such *state management* sentences (§6.8.6 in RFC 5880). The RFC contains additional components that SAGE currently can not handle. These include algorithms (*e.g.,* timing calculation) and complex communication patterns (*e.g.,* authentication). In §7, we discuss what it will take to extend SAGE to completely parse BFD.

BFD Introduction. BFD is used to detect faults between two nodes. Each node maintains multiple state variables for both protocol and connection state. Connection state is represented by a 3-state machine and represents the status (*e.g.,* established, being established, or being torn down) of the session between nodes. Protocol state variables are used to track local and remote configuration.[9]

State Management Dictionary. A state management sentence describes how to use or modify protocol or connection state in terms of state management variables. For example, *bfd.SessionState* is a connection state variable; *Up* is a permitted value. We extend our term dictionary to include these state variables and values as noun phrases.

Parsing. We focus on explaining our analysis of such state management sentences. SAGE is also able to parse the BFD packet header described in §4.1 of RFC 5880. We analyzed 22 state management sentences in §6.8.6 of RFC 5880 which involve a greater diversity of operations than pure packet generation. To support these, we added 15 lexical entries, 10 predicates, and 8 function handlers.

[9]This is common across protocols: for example, TCP keeps track of protocol state regarding ACK reception.

Category	Example	Count
More than 1 LF	To form an echo reply message, the source and destination addresses are simply reversed, the type code changed to 0, and the checksum recomputed.	4
0 LF	Address of the gateway to which traffic for the network specified in the internet destination network field of the original datagram's data should be sent.	1
Imprecise sentence	If code = 0, an identifier to aid in matching echos and replies, may be zero.	6

Table 5: Examples of categorized rewritten text.

6.5 Disambiguation

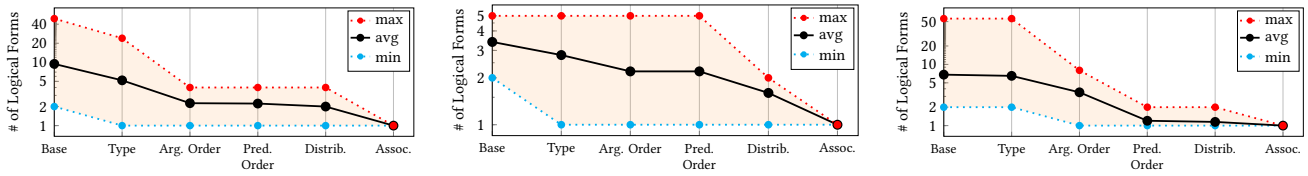
Revising a specification inevitably requires some degree of manual inspection and disambiguation. SAGE makes this systematic: it identifies and fixes ambiguities when it can, alerts specification authors or developers when it cannot, and can help iteratively verify re-written parts of the specification.

Ambiguous sentences. When we began to analyze RFC 792 with SAGE, we immediately found many ambiguities we highlighted throughout this paper; these result in more than one logical form even after manual disambiguation.

We also encountered ostensibly disambiguated text that yields zero logical forms; this is caused by incomplete sentences. For example, "If code = 0, identifies the octet where an error was detected" fails CCG parsing due to lack of subject in the sentence, and indeed it may not be parseable for a human lacking context regarding the referent. Such sentence fragments require human guesswork, but, as we have observed in §4, we can leverage structural context in the RFC in cases where the referent of these sentences is a field name. In these cases, SAGE is able to correctly parse the sentence by supplying the parser with the subject.

Among 87 instances in RFC 792, we found 4 that result in more than 1 logical form and 1 results in 0 logical forms (Table 5). We rewrote these 5 ambiguous (of which only 3 are unique) sentences to enable automated protocol generation. These ambiguous sentences were found after SAGE had applied its checks (§4.2)—these are in a sense true ambiguities in the ICMP RFC. In SAGE, we require the user to revise such sentences, according to the feedback loop as shown in Figure 2. SAGE keeps the resulting LFs from an ambiguous sentence after applying the disambiguation checks; comparing these LFs can help users identify where the ambiguity lies, thus guiding their revisions. In our end-to-end experiments (§6.2), we evaluated SAGE using the modified RFC with these ambiguities fixed.

Under-specified behavior. SAGE can also discover under-specified behavior through unit testing; generated code can be applied to unit tests to see if the protocol implementation is complete. In this process, we discovered 6 sentences that are variants of this sentence: "If code = 0, an identifier to aid in matching echos and replies, may be zero". This sentence does not specify whether the sender or the receiver or both can (potentially) set the identifier. The correct behavior is only for the sender to follow this instruction; a sender may generate a non-zero identifier,



(a) ICMP (b) IGMP (c) BFD

Figure 5: Number of LFs after Inconsistency Checks on ICMP/IGMP/BFD text: for each ambiguous sentence, sequentially executing checks on LFs (Base) reduces inconsistencies; after the last Associativity check, the final output is a single LF.

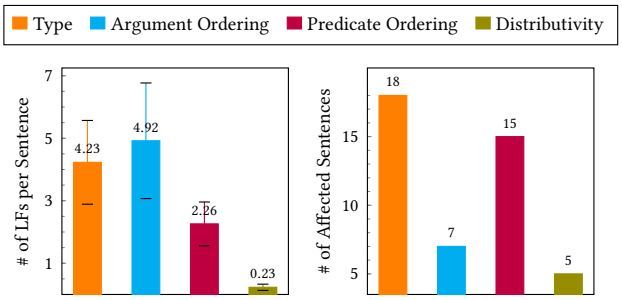


Figure 6: Effect of individual disambiguation checks on RFC 792: *Left:* average number of LFs filtered by the check per ambiguous sentence with standard error *Right:* number of ambiguous sentences affected out of 42 total.

and the receiver should set the identifier to be zero in the reply. Not doing so results in a non-interoperability with Linux's *ping* implementation.

Efficacy of logical form winnowing. SAGE winnows logical forms so it can automatically disambiguate text when possible, reducing manual labor in disambiguation. To show why winnowing is necessary, and how effective each of its checks can be, we collect text fragments that could lead to multiple logical forms, and calculate how many are generated before and after we perform inconsistency checks along with the isomorphism check. We show the extent to which each check is effective in reducing logical forms: in Figure 5a, the max line shows the description that leads to the highest count of generated logical forms and shows how the value goes down to one after all checks are completed. Similarly, the min line represents the situation for the text that generates the fewest logical forms before applying checks. Between the min and max lines, we also show the average trend among all sentences.

Figure 5a shows that all sentences resulted in 2-46 LFs, but SAGE's winnowing reduces this to 1 (after human-in-the-loop rewriting of true ambiguities). Of these, type, argument ordering and the associativity checks are the most effective. We apply the same analysis to IGMP (Figure 5b). In IGMP, the distributivity check is also important. This analysis shows the cumulative effect of applying checks in the order shown in the figure. We also apply the same analysis to BFD state management sentences (Figure 5c). We discover some longer sentences could result in up to 56 LFs.

A more direct way to understand the efficacy of checks is shown in Figure 6 (for ICMP). To generate this figure, for each sentence, we apply only one check on the base set of logical forms and measure how many LFs the check can reduce. The graphs show the mean and standard deviation of this number across sentences, and the number of sentences to which a check applies. For ICMP, as before, type and predicate ordering checks reduced LFs for the most number of sentences, but argument ordering reduced the most logical forms. For IGMP (omitted for brevity), the distributivity checks were also effective, reducing one LF every 2 sentences.

Figure 5 does not include NTP; for the parts of this RFC that SAGE analyzes, the base semantic parser produces *at most 2 LFs* (after adding a small number of lexical entries and checks §6.3), and the additional checks winnow these down to 1 LF.

Sentence	Label	#LFs
The 'address' of the 'source' in an 'echo message' will be the 'destination' of the 'echo reply' 'message'.	Poor	16
The 'address' of the 'source' in an 'echo message' will be the 'destination' of the 'echo reply message'.	Good	6

Table 6: Comparison of the number of logical forms (LFs) between good and poor noun phrase labels.

	Increase	Decrease	Zero
Domain-specific Dict.	17	0	0
Noun-phrase Labeling	0	8	54

Table 7: Effect of disabling domain-specific dictionary and noun-phrase labeling on number of logical forms.

Importance of Noun Phrase Labeling. SAGE requires careful labeling of noun-phrases using SpaCy based on a domain-specific dictionary (§3). This is an important step that can significantly reduce the number of LFs for a sentence. To understand why, consider the example in Table 6, which shows two different noun-phrase labels, which differ in the way SAGE labels the fragment "echo reply message". When the entire fragment is not labeled as a single noun phrase, CCG outputs many more logical forms, making it harder to disambiguate the sentence. In the limit, when SAGE does not use careful noun phrase labeling, CCG is unable to parse some sentences at all (resulting in 0 LFs).

Table 7 quantifies the importance of these components. Removing the domain-specific dictionary increases the number of logical forms (before winnowing) for 17 of the 87 sentences in the ICMP RFC. Completely removing noun-phrase labeling using SpaCy has more serious consequences: 54 sentences result in 0 LF. Eight other sentences result in fewer LFs, but these reduce to 0 after winnowing.

7 SAGE LIMITATIONS

While SAGE takes a significant step toward automated specification processing, much work remains.

Specification components. To understand this, we have manually inspected several protocol specifications and categorized components of specifications into two categories: syntactic and conceptual.

	IPv4	TCP	UDP	ICMP	NTP	OSPF2	BGP4	RTP	BFD
♦ Packet Format	x	x	x	x	x	x	x	x	x
♦ Interoperation	x	x	x	x	x	x	x	x	
♦ Pseudo Code	x	x	x	x	x	x	x	x	x
+ State/Session Mngmt.		x				x	x	x	x
Comm. Patterns	x	x				x	x	x	x
Architecture					x	x		x	

Table 8: Conceptual components in RFCs. SAGE supports components marked with ♦ (fully) and + (partially).

	IPv4	TCP	UDP	ICMP	NTP	OSPF2	BGP4	RTP	BFD
♦ Header Diagram	x	x	x	x	x	x	x	x	x
♦ Listing	x	x	x	x	x	x	x	x	x
Table	x	x			x	x	x	x	x
Algorithm Description	x	x			x	x		x	x
Other Figures	x				x	x	x	x	
Seq./Comm. Diagram	x	x			x	x		x	
State Machine Diagram		x							x

Table 9: Syntactic components in RFCs. SAGE supports parsing the syntax of those marked with ♦ (fully).

SENTENCE	The timeout procedure is called in client mode and symmetric mode when the peer timer reaches the value of the timer threshold variable.

CODE	`if (peer.timer >= peer.threshold) {`
	`  if (symmetric_mode || client_mode) {`
	`  timeout_procedure();`
	`  }`
	`}`

Table 10: NTP peer variable sentence and resulting code.

Conceptual components (Table 8) describe protocol structure and behavior: these include header field semantic descriptions, specification of sender and receiver behavior, who should communicate with whom, how sessions should be managed, and how protocol implementations should be architected.

RFC authors augment conceptual text with syntactic components (Table 9). These include forms that provide better understanding of a given idea (*e.g.,* header diagrams, tables, state machine descriptions, communication diagrams, and algorithm descriptions). SAGE includes support for two of these elements; adding support for others is not conceptually difficult, but may require significant programming effort.

Conceptual components may require significant additional research. Most popular standards have many, if not all, of these elements. SAGE supports parsing of 3 of the 6 conceptual elements in Table 8, for ICMP and parts of IGMP, NTP, and BFD. Our results (§6.2) show that extending these elements to other protocols can, in some cases, require marginal extensions at each step. In addition, SAGE is already able to parse state management for some protocols. However, much work remains to achieve complete generality, of which state and session management is a significant piece.

BFD state management. When we performed CCG parsing and code generation on state management sentences, we found two types of sentences that could not be parsed correctly (Table 11). Both of these sentences reveal limitations in the underlying NLP approach we use.

The CCG parser treats each sentence independently, but the first example in Table 11 illustrates dependencies across sentences.

TYPE		EXAMPLE
Nested code	Original	If the Your Discriminator field is nonzero, it MUST be used to select *the session* with which this BFD packet is associated. If *no session* is found, the packet MUST be discarded.
	Rewritten	If the Your Discriminator field is nonzero, it MUST be used to select the session with which this BFD packet is associated. If *the Your Discriminator field is nonzero and* no session is found, the packet MUST be discarded.
Rephrasing	Original	If *bfd.RemoteDemandMode is 1*, bfd.SessionState is Up, and bfd.RemoteSessionState is Up, *Demand mode is active on the remote system* and the local system MUST cease the periodic transmission of BFD Control packets.
	Rewritten	If bfd.RemoteDemandMode is 1, bfd.SessionState is Up, and bfd.RemoteSessionState is Up, the local system MUST cease the periodic transmission of BFD Control packets.

Table 11: Challenging BFD state management sentences.

Specifically, SAGE must infer that the reference to *no session* in the second sentence must be matched to *the session* in the first sentence. This is an instance of the general problem of co-reference resolution [27], which can resolve identical noun phrases across sentences. To our knowledge, semantic parsers cannot yet resolve such references. To get SAGE to parse the text, we rewrote the second sentence to clarify the co-reference, as shown in Table 11.

The second sentence contains three conditionals, followed a non-actionable fragment that rephrases one of the conditionals. Specifically, the first condition *if bfd.RemoteDemandMode is 1*, is rephrased, in English, immediately afterwards (*Demand mode is active on the remote node*). To our knowledge, current NLP techniques cannot easily identify rephrased sentence fragments. SAGE relies on human annotation to identify this fragment as non-actionable; after removing the fragment, it is able to generate code correctly for this sentence.

NTP state management. The NTP RFC has complex sentences on maintaining peer and system variables, to decide when each procedure should be called and when variables should be updated. One example sentence, shown in Table 10, concerns when to trigger timeout. SAGE is able to parse the sentence into an LF and turn it into a code snippet. However, NTP requires more complex co-reference resolution, as other protocols may too [27, 31]: in NTP, context for state management is spread throughout the RFC and SAGE will need to associate these conceptual references. For instance, the word "and" in the example (Table 10) could be equivalent to a logical AND *or* a logical OR operator depending on whether symmetric mode and client mode are mutually exclusive or not. A separate section clarifies that the correct semantics is OR.

Reducing Human Effort. An important direction for future work is to minimize the manual effort currently required for disambiguation and code generation. Our winnowing reduces the number of instances where users have to supply new lexical entries or checks (§4.2); we cannot quantify the number of such new entries required for RFC text we have yet to examine, but expect that it will decrease over time as more protocols are supported and more entries are in SAGE's entry database. We also cannot state definitively the generality of our current code generation approach. When users have to intervene, SAGE reduces cognitive load for the user by suggesting possible lexical entry additions. Future work will need to explore similar usability enhancements for other human input tasks: adding

new predicate checks, specifying cross-references (references to other protocols in a specification), and identifying non-actionable sentences. Future work can also explore tools that automate some of these steps entirely (*e.g.,* identifying cross-references, or identifying non-actionable sentences) as well as techniques that improve the readability of generated code. Finally, we have attempted to make SAGE's auto-generated code clear by ensuring that we adopt naming conventions for variables from RFCs and automatically emit context (*i.e.,* add an original sentence from an RFC as a comment) for each snippet of generated code. Future work can explore how to auto-generate truly elegant code.

8 RELATED WORK

Protocol Languages / Formal Specification Techniques. Numerous protocol languages have been proposed over the years. In the '80s, Estelle [14] and LOTOS [12] provided formal descriptions for OSI protocol suites. Although these formal techniques can specify precise protocol behavior, it is hard for people to understand and thus use for specification or implementation. Estelle used finite state machine specifications to depict how protocols communicate in parallel, passing on complexity, unreadability, and rigidity to followup work [13, 70, 80]. Other research such as RTAG [3], x-kernel [30], Morpheus [1], Prolac [40], Network Packet Representation [53], and NCT [52] gradually improved readability, structure, and performance of protocols, spanning specification, testing, and implementation. However, we find and the networking community has found through experience, that English-language specifications are more readable than such protocol languages.

Protocol Analysis. Past research [10–12] developed techniques to reason about protocol behaviors in an effort to minimize bugs. Such techniques used finite state machines, higher-order logic, or domain-specific languages to verify protocols. Another thread of work [38, 39, 45] explored the use of explicit-state model-checkers to find bugs in protocol implementations. This thread also inspired work (*e.g.,* [59]) on discovering non-interoperabilities in protocol implementations. While our aims are similar, our focus is end-to-end, from specification to implementation, and on identifying where specification ambiguity leads to bugs.

NLP for Log Mining and Parsing. Log mining and parsing are techniques that leverage log files to discover and classify different system events (e.g., 'information', 'warning', and 'error'). Past studies have explored Principal Component Analysis [84], rule-based analysis [25], statistic analysis [56, 79], and ML-based methods [72] to solve log analysis problems. Recent work [6, 9] has applied NLP to extract semantic meanings from log files for event categorization. SAGE is complementary to this line of work: while it uses NLP to categorize sender/receiver roles, SAGE takes the additional step of generating code.

Program Synthesis. To automatically generate code, prior work has explored program synthesis. Reactive synthesis [61, 62] relies on interaction with users to read input for generating output programs. Inductive synthesis [2] recursively learns logic or functions with incomplete specifications. Proof-based synthesis (*e.g.,* [73]) takes a correct-by-construction approach to develop inductive proofs to extract programs. Type-based synthesis [24, 58] takes advantage of the types provided in specifications to refine output. In networking, program synthesis techniques can automate (*e.g.,* [50, 51]) updating of network configurations, and generating programmable switch code [26]. It may be possible to use program synthesis in SAGE to generate protocol fragments.

Semantic Parsing and Code Generation. Semantic parsing is a fundamental task in NLP that aims to transform unstructured text into structured LFs for subsequent execution [8]. For example, to answer the question "Which team does Frank Hoffman play for?", a semantic parser generates a structured query "SELECT TEAM from table where PLAYER=Frank Hoffman" with SQL Standard Grammar [18]. A SQL interpreter can execute this query on a database and give the correct answer [34]. Apart from the application to question answering, semantic parsing has also been successful in navigating robots [77], understanding instructions [15], and playing language games [81]. Research in generating code from natural language goes beyond LFs, to output concrete implementations in high-level general-purpose programming languages [48]. This problem is usually formulated as syntax-constrained sequence generation [46, 87]. The two topics are closely related to our work since the process of implementing network protocols from RFCs requires the ability to understand and execute instructions.

Pre-trained Language Models. Recently, high-capacity pre-trained language models [20, 43, 60, 85] have dramatically improved NLP in question answering, natural language inference, text classification, *etc.* The general approach is to first train a model on a huge corpus with unsupervised learning (*i.e.,* pre-training), then re-use these weights to initialize a task-specific model that is later trained with labeled data (*i.e.,*, fine-tuning). In the context of SAGE, such pre-trained models advance improve semantic parsing [89, 90]. Recent work [23] also attempts to pre-train on programming and natural languages simultaneously, and achieves state-of-the-art performance in code search and code documentation generation. However, direct code generation using pre-trained language models is an open research area and requires massive datasets; the best model for a related problem, natural language generation, GPT [65], requires 8 M web pages for training.

9 CONCLUSIONS

This paper describes SAGE, which introduces semi-automated protocol processing across multiple protocol specifications. SAGE includes domain-specific extensions to semantic parsing and automated discovery of ambiguities and enables disambiguation; SAGE can convert these specifications to code. Future work can extend SAGE to parse more specification elements, and devise better methods to involve humans in the loop to detect and fix ambiguities and guide the search for bugs.

Acknowledgements. We thank our shepherd Noa Zilberman, the anonymous reviewers, and the artifact evaluation committee for their feedback. This paper was supported in part by the U.S. National Science Foundation (CNS-1901523 and IIS-2048211), and by an Annenberg Fellowship at USC.

REFERENCES

[1] Abbott, M. B., and Peterson, L. L. A language-based approach to protocol implementation. *IEEE/ACM transactions on networking* (1993).

[2] Alur, R., Bodik, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madusudan, P., Martin, M., Raghothman, M., et al. Syntax-guided synthesis. dependable software systems engineering. *NATO Science for Peace and Security Series (2014). http://sygus. seas. upenn. edu/files/sygus_extended. pdf* (2014).

[3] Anderson, D. P. Automated protocol implementation with rtag. *IEEE Transactions on Software Engineering 14*, 3 (1988), 291–300.

[4] Artzi, Y. Cornell SPF: Cornell Semantic Parsing Framework, 2016.

[5] Artzi, Y., FitzGerald, N., and Zettlemoyer, L. S. Semantic parsing with combinatory categorial grammars. *ACL (Tutorial Abstracts) 3* (2013).

[6] Aussel, N., Petetin, Y., and Chabridon, S. Improving performances of log mining for anomaly prediction through nlp-based log parsing. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2018), IEEE, pp. 237–243.

[7] Beckett, R., Mahajan, R., Millstein, T., Padhye, J., and Walker, D. Network configuration synthesis with abstract topologies. *SIGPLAN Not. 52*, 6 (June 2017), 437–451.

[8] Berant, J., Chou, A., Frostig, R., and Liang, P. Semantic parsing on freebase from question-answer pairs. In *Proceedings of the 2013 conference on empirical methods in natural language processing* (2013), pp. 1533–1544.

[9] Bertero, C., Roy, M., Sauvanaud, C., and Trédan, G. Experience report: Log mining using natural language processing and application to anomaly detection. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)* (2017), IEEE, pp. 351–360.

[10] Bhargavan, K., Obradovic, D., and Gunter, C. A. Formal verification of standards for distance vector routing protocols. *Journal of the ACM (JACM) 49*, 4 (2002), 538–576.

[11] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications* (2005), pp. 265–276.

[12] Bolognesi, T., and Brinksma, E. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems 14*, 1 (1987).

[13] Boussinot, F., and De Simone, R. The esterel language. *Proceedings of the IEEE 79*, 9 (1991), 1293–1304.

[14] Budkowski, S., and Dembinski, P. An introduction to estelle: a specification language for distributed systems. *Computer Networks and ISDN systems 14*, 1 (1987), 3–23.

[15] Chen, D. L., and Mooney, R. J. Learning to interpret natural language navigation instructions from observations. In *Twenty-Fifth AAAI Conference on Artificial Intelligence* (2011).

[16] Clark, D. D. A cloudy crystal ball: visions of the future. *Proceedings of the Twenty-Fourth Internet Engineering Task Force* (1992), 539–544.

[17] D. Harkins, E. Secure Password Ciphersuites for Transport Layer Security (TLS). RFC 8492, 2019.

[18] Date, C. J. *A Guide to the SQL Standard: A User's Guide to the Standard Relational Language SQL*. Addison-Wesley Longman Publishing Co., Inc., USA, 1987.

[19] Deering, D. S. E. Host extensions for IP multicasting. RFC 1112, 1989.

[20] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[21] Dong, L., and Lapata, M. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793* (2018).

[22] Editor, R., and Flanagan, H. RFC Style Guide. RFC 7322, Sept. 2014.

[23] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. *ArXiv abs/2002.08155* (2020).

[24] Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices 50*, 6 (2015), 229–239.

[25] Fu, Q., Lou, J.-G., Wang, Y., and Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining* (2009), IEEE, pp. 149–158.

[26] Gao, X., Kim, T., Wong, M. D., Raghunathan, D., Varma, A. K., Kannan, P. G., Sivaraman, A., Narayana, S., and Gupta, A. Switch code generation using program synthesis. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, p. 44–61.

[27] Group, S. N. Corenlp coreference resolution. https://stanfordnlp.github.io/CoreNLP/coref.html.

[28] Hockenmaier, J., and Bisk, Y. Normal-form parsing for combinatory categorial grammars with generalized composition and type-raising. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)* (Beijing, China, Aug. 2010), Coling 2010 Organizing Committee, pp. 465–473.

[29] Honnibal, M., and Montani, I. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[30] Hutchinson, N. C., and Peterson, L. L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software engineering*, 1 (1991), 64–76.

[31] Institute, A. A. AllenNLP Coreference Resolution. https://demo.allennlp.org/coreference-resolution.

[32] Ipp interoperability testing event #2. http://www.pwg.org/ipp/testing/bake2.html.

[33] Jethanandani, M., Agarwal, S., Huang, L., and Blair, D. YANG Data Model for Network Access Control Lists (ACLs). RFC 8519, 2019.

[34] Kamath, A., and Das, R. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).

[35] Katz, D., and Ward, D. Bidirectional Forwarding Detection (BFD). RFC 5880, 2010.

[36] Kempson, R. M., and Cormack, A. Ambiguity and quantification. *Linguistics and Philosophy 4*, 2 (1981), 259–309.

[37] Kessens, D., Bates, T. J., Alaettinoglu, C., Meyer, D., Villamizar, C., Terpstra, M., Karrenberg, D., and Gerich, E. P. Routing Policy Specification Language (RPSL). RFC 2622, June 1999.

[38] Killian, C., Anderson, J. W., Jhala, R., and Vahdat, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)* (2007), NSDI, USENIX Association.

[39] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: language support for building distributed systems. *ACM SIGPLAN Notices 42*, 6 (2007), 179–188.

[40] Kohler, E., Kaashoek, M. F., and Montgomery, D. R. A readable tcp in the prolac protocol language. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (1999), pp. 3–13.

[41] Krishnamurthy, J., Dasigi, P., and Gardner, M. Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing* (2017), pp. 1516–1526.

[42] Kurose, J., and Ross, K. Computer networking: A top down approach, 2012.

[43] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[44] Lantz, B., Heller, B., and McKeown, N. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (2010), pp. 1–6.

[45] Lee, H., Seibert, J., Killian, C. E., and Nita-Rotaru, C. Gatling: Automatic attack discovery in large-scale distributed systems. In *NDSS* (2012).

[46] Liang, C., Berant, J., Le, Q., Forbus, K. D., and Lao, N. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020* (2016).

[47] Lin, X. V., Wang, C., Pang, D., Vu, K., and Ernst, M. D. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01* (2017).

[48] Ling, W., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., Senior, A., Wang, F., and Blunsom, P. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[49] Loper, E., and Bird, S. Nltk: the natural language toolkit. *arXiv preprint cs/0205028* (2002).

[50] McClurg, J., Hojjat, H., Černỳ, P., and Foster, N. Efficient synthesis of network updates. *Acm Sigplan Notices 50*, 6 (2015), 196–207.

[51] McClurg, J., Hojjat, H., Foster, N., and Černỳ, P. Event-driven network programming. *ACM SIGPLAN Notices 51*, 6 (2016), 369–385.

[52] McMillan, K. L., and Zuck, L. D. Formal specification and testing of QUIC. In *Proceedings of ACM SIGCOMM* (2019).

[53] McQuistin, S., Band, V., Jacob, D., and Perkins, C. Parsing protocol standards to parse standard protocols. In *Proceedings of the Applied Networking Research Workshop* (New York, NY, USA, 2020), ANRW '20, Association for Computing Machinery, p. 25–31.

[54] Mills, D. Network Time Protocol (version 1) specification and implementation. RFC 1059, 1988.

[55] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software defined networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 1–13.

[56] Nagaraj, K., Killian, C., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 353–366.

[57] List of NLTK dependents. https://github.com/nltk/nltk/network/dependents.

[58] Osera, P.-M., and Zdancewic, S. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices 50,* 6 (2015), 619–630.
[59] Pedrosa, L., Fogel, A., Kothari, N., Govindan, R., Mahajan, R., and Millstein, T. Analyzing protocol implementations for interoperability. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 485–498.
[60] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365* (2018).
[61] Piterman, N., Pnueli, A., and Sa'ar, Y. Synthesis of reactive (1) designs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation* (2006), Springer, pp. 364–380.
[62] Pnueli, A., and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), pp. 179–190.
[63] Postel, J. Internet Control Message Protocol. RFC 792, 1981.
[64] Rabinovich, M., Stern, M., and Klein, D. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).
[65] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI Blog 1,* 8 (2019), 9.
[66] Rayner, K., and Duffy, S. A. Lexical complexity and fixation times in reading: Effects of word frequency, verb complexity, and lexical ambiguity. *Memory & cognition 14,* 3 (1986), 191–201.
[67] Rfc editor. http://www.rfc-editor.org/.
[68] S. Gueron, A. Langley, Y. L. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. RFC 8452, 2019.
[69] SAGE. https://github.com/USC-NSL/sage.
[70] Sidhu, D., and Chung, A. *A formal description technique for protocol engineering.* University of Maryland at College Park, 1990.
[71] First sip interoperability test event. https://www.cs.columbia.edu/sip/sipit/1/, 2008.
[72] Sipos, R., Fradkin, D., Moerchen, F., and Wang, Z. Log-based predictive maintenance. In *Proceedings of the 20th ACM SIGKDD international conference on knowledge discovery and data mining* (2014), pp. 1867–1876.
[73] Srivastava, S., Gulwani, S., and Foster, J. S. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2010), pp. 313–326.
[74] Srivastava, S., Labutov, I., and Mitchell, T. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 conference on empirical methods in natural language processing* (2017), pp. 1527–1536.
[75] Steedman, M., and Baldridge, J. Combinatory categorial grammar. *Non-Transformational Syntax: Formal and explicit models of grammar* (2011), 181–224.
[76] Tcpdump & libpcap public repository. https://www.tcpdump.org/. Accessed: 2020-05-22.
[77] Tellex, S., Kollar, T., Dickerson, S., Walter, M. R., Banerjee, A. G., Teller, S., and Roy, N. Understanding natural language commands for robotic navigation and mobile manipulation. In *Twenty-fifth AAAI conference on artificial intelligence* (2011).
[78] Thomson, M. Example Handshake Traces for TLS 1.3. RFC 8448, 2019.
[79] Vaarandi, R. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)* (2003), IEEE, pp. 119–126.
[80] von Bochmann, G. *Methods and tools for the design and validation of protocol specifications and implementations.* Université de Montréal, Département d'informatique et de recherche ..., 1987.
[81] Wang, S. I., Liang, P., and Manning, C. D. Learning language games through interaction. *arXiv preprint arXiv:1606.02447* (2016).
[82] Wang, Z., Qin, Y., Zhou, W., Yan, J., Ye, Q., Neves, L., Liu, Z., and Ren, X. Learning from explanations with neural execution tree. In *International Conference on Learning Representations* (2020).
[83] White, M., and Rajkumar, R. A more precise analysis of punctuation for broad-coverage surface realization with ccg. In *Coling 2008: Proceedings of the workshop on Grammar Engineering Across Frameworks* (2008), pp. 17–24.
[84] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 117–132.
[85] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems* (2019), pp. 5754–5764.
[86] Yin, P., and Neubig, G. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).
[87] Yin, P., and Neubig, G. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Vancouver, Canada, July 2017), Association for Computational Linguistics.
[88] Yin, P., Zhou, C., He, J., and Neubig, G. Structvae: Tree-structured latent variable models for semi-supervised semantic parsing. *arXiv preprint arXiv:1806.07832* (2018).
[89] Zhang, S., Ma, X., Duh, K., and Durme, B. V. Amr parsing as sequence-to-graph transduction. *ArXiv abs/1905.08704* (2019).
[90] Zhang, S., Ma, X., Duh, K., and Durme, B. V. Broad-coverage semantic parsing as transduction. In *EMNLP/IJCNLP* (2019).

APPENDIX

A ICMP Test Scenario Setup

Destination Unreachable Message. At the router/receiver side, we assume the router only recognizes three subnets, which are 10.0.1.1/24, 192.168.2.1/24, and 172.64.3.1/24. At the sender side, we craft the packet with destination IP address not belonging to any of the three subnets. The receiver reads the packet and calls the generated function to construct the destination unreachable message back to the sender.

Time Exceeded Message. At the sender side, we intentionally generate a packet with the time-to-live field in IP header set to 1, and the destination IP address set to server 1's address. At the router side, the router checks the value of time-to-live field and recognizes the packet cannot reach the destination before the time-to-live field counts down to zero. The router interface calls the generated function to construct a time exceed message and sends it back to the client.

Parameter Problem Message. At the router side, we assume the router can only handle IP packets in which the type of service value equals to zero. At the sender side, we modify the sent packet to set the type of service value to one. The router interface recognizes the unsupported type of service value and calls the generated function to construct a parameter problem message back to the client.

Source Quench Message. At the receiver side, we assume one outbound buffer is full, and therefore there is no space to hold a new datagram. At the sender side, we generated a packet to server 1. If there is still buffer space for the router to forward the packet to server 1, the router should push the packet to the outbound buffer connected to the subnet where server 1 belongs to. Under this scenario, the router will decide to discard the received packet, and construct a source quench packet back to the client.

Redirect Message. At the sender side, the client generated a packet to an IP address that is within the same subnet, but sent to the router. The router discovered the next gateway is in the same subnet as the sender host, and therefore constructs the redirect message to the client with the redirect gateway address by calling the generated functions.

Echo and Echo Reply Message. In RFC 792, echo/echo reply are explained together, but some sentences are merely for echo while some are only for echo reply. After analysis, SAGE generates two different pieces of code. One is specific to the sender side, and the other is specific to the receiver side. The client calls the generated function to construct an echo message to the router interface. The router interface finds it is the destination and constructs an echo reply message back to the client by calling the receiver code.

Timestamp and Timestamp Reply Message. The sender and receiver behavior in this scenario is identical to echo/echo reply. The sender sends a packet by calling the generated function and

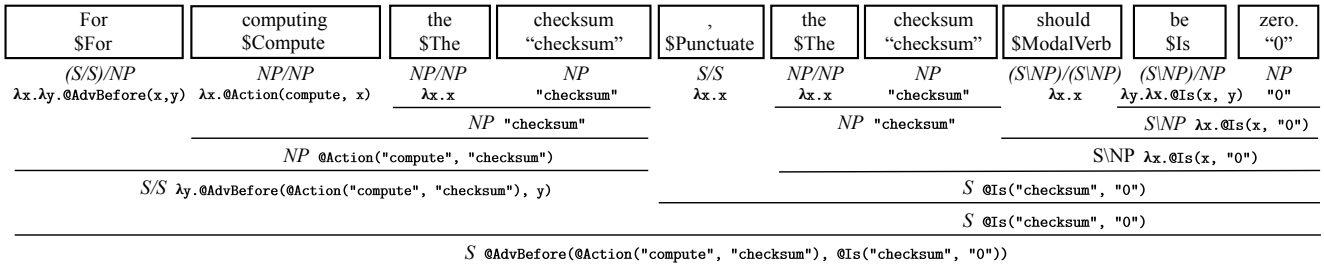


Figure 7: Constructing one logical form of sentence *"For computing the checksum, the checksum should be zero"* with CCG.

the receiver matches the ICMP type and replies to packets with the generated function. The difference lies in the packet generated by the function. The timestamp or timestamp reply message do not have datagram data, but they have three different timestamp fields in its header. The generated function correctly separates three different timestamps with respect to the roles and computation time.

Information Request and Reply Message. The sender and receiver behavior of this scenario is the same as echo/echo reply and timestamp/timestamp reply. Similar to timestamp/timestamp reply, the differences lie in the generated packets that do not have data; the field values are different.

Interoperation with existing tools. To test whether a SAGE-generated ICMP implementation interoperates with tools like **`ping`** and **`traceroute`**, we integrated our static framework code and the SAGE-generated code into a Mininet-based framework used for the course described in §2. With this framework, we verified, with four Linux commands (testing echo, destination unreachable, time exceeded, and traceroute behavior) shown in Table 12, that a SAGE-generated receiver or router correctly processes echo request packets sent and received by built-in **`ping`** and **`traceroute`**.

Test Command	Purpose
`client ping -c 10 10.0.1.1`	Test echo msg
`client ping -c 10 192.168.3.1`	Test dest unreachable msg
`client ping -c 10 -t 1 192.168.2.2`	Test time exceeded msg
`client traceroute -I 10.0.1.1`	Test traceroute

Table 12: ICMP test commands used in project environment.

B CCG Parsing Example

We show a more complex example, of deriving one final logical form from the sentence: "For computing the checksum, the checksum should be zero." in Figure 7. First, each word in the sentence is mapped to its lexical entries (*e.g.*, checksum → NP: "checksum"). Multiple lexical entries may be available for one word; in this case we make multiple attempts to parse the whole sentence with each entry. After this step, the CCG parsing algorithm automatically applies combination rules and derives final logical forms for the whole sentence.