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 over the years protocol implementations exhibited bugs and non-interoperabilities. In this paper, we explore to what extent natural language processing (NLP), an area that has made impressive strides in recent years, can be used to generate protocol implementations. We advocate a semi-automated protocol generation approach, SAGE, that can be used to uncover ambiguous or under-specified sentences in specifications; these can then be fixed by a human iteratively until SAGE is able to generate protocol code automatically. Using an implementation of SAGE, we discover 5 instances of ambiguity and 6 instances of under-specification in the ICMP RFC, after fixing which SAGE is able to generate code automatically that interoperates perfectly with Linux implementations. We demonstrate the ability to generalize SAGE to parts of IGMP and NTP. We also find that SAGE supports half of the conceptual components found in major standards protocols; this suggests that, with some additional machinery, SAGE may be able to generalize to TCP and BGP.

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 [15]. In that time we have come to depend far more on network protocols than most imagined at the outset. To this day, engineers implement a network protocol by reading and interpreting protocol specifications as described in Request For Comments documents (RFCs). Their challenge is to navigate easy-to-misinterpret colloquial language while producing not only a bug-free implementation but also one that *interoperates* with another implementation written by another engineer at a different time and place.

Software engineers find it difficult to interpret today's protocol specifications in large part because natural language can be imprecise and introduce ambiguity. Unfortunately, such ambiguity is not rare; the errata alone for RFCs over the years highlight numerous ambiguities and the problems they have raised [16, 32, 64, 73]. Ambiguity has resulted in buggy implementations and expensive and time-consuming software engineering processes, like interoperability bake-offs [31, 66], to discover the root causes for non-interoperability between different implementations of the same protocol.

To address this, one line of research has advocated for the formal specification of programs and protocols (§8), which would enable verifying the correctness of specifications and, potentially, automatically translating the spec to running code [12]. However, formal specifications are cumbersome and thus have not been adopted in practice; to date, protocols are still specified in natural language. (In recent years, attempts have been made to formalize other aspects of network operation, such as network configuration [6, 35] and control plane behavior [52], with varying degrees of success.)

In this paper, we consider whether natural language processing (NLP) might enable automated generation of protocol implementations from specification text. NLP has made impressive strides in recent years; we explore to what degree it is now possible to generate protocol code from English text.

Our key challenge is to understand the *semantics* of a natural language specification. This task, called *semantic parsing*, has seen much work in recent years, including the development of reasonably mature parsing tools (*e.g.,* CCG [4]). These tools usually describe natural language with a *lexicon*, and generate a semantic interpretation for each sentence in a text. Because they are trained on generic prose, they cannot be expected to work out of the box for network protocol specifications, which contain idiomatic use such as syntactic cues often embedded in RFCs (such as structured descriptions of fields) and incomplete sentences, and assume context from neighboring sentences or from other protocols. More importantly, the richness of natural language will likely always result in ambiguity, and at least for the foreseeable future, we do not expect NLP-based systems to work in a fully automated fashion (§2).

Contributions. In this paper, we advocate for a *semi-automated* approach to protocol generation from natural language, called SAGE, which works in an iterative manner. SAGE reads the text of an RFC and iteratively marks sentences (a) for which it cannot generate unique semantic interpretations or (b) which fail on *unit tests* designed for the protocol (SAGE leverages test-driven development). Sentences of the former type are likely semantically ambiguous, and sentences of the latter type represent under-specified behaviors. In either case, the spec author is given the choice of re-writing the sentences (perhaps over several iterations), until the resulting modified RFC can cleanly generate code.

To this end, we make the following contributions. We tailor (§3) a pre-existing semantic parser to extend its vocabulary and its lexicon to cover ways in which RFCs use natural language in a domain-specific manner. Our extensions also extract structural context from RFC documents that may be crucial to understanding a sentence's semantics and to generate code. Even so, such a parser can *fail* to

extract a sentence's semantics, or emit *multiple* semantic interpretations; ideally, the parser should generate exactly one semantic interpretation.

Our second contribution is to observe that many of these non-idealities occur because of idiomatic conventions used in RFCs. RFCs use grammatically incomplete sentences to describe protocol header fields, and use verbs like *is* and prepositions like *of* in somewhat specific ways. Using these conventions, we devise automated techniques to eliminate some of the semantic interpretations. If, after this *disambiguation* step (§4), a sentence has multiple interpretations, it is likely to be fundamentally ambiguous, so SAGE requires human intervention to rewrite the sentence.

Our third contribution is a *code generator* that converts semantic representations to executable code (§5). The parser represents a sentence's semantics using logical predicates, and the code generator emits executable code, using contextual information that it has gleaned from the RFC's document structure, as well as static context pre-defined in SAGE about lower-layer protocols and the underlying OS. Unit testing on generated code can uncover incompleteness in specifications.

We have developed a 3-stage pipeline embodying these contributions into a system also called SAGE. SAGE discovered (§6) *5* sentences in the ICMP RFC [59] (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 of which are variants of a single sentence). After fixing these, SAGE was able to automatically generate an implementation of ICMP that interoperated perfectly with Linux's implementations of `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 significant sections of another RFC, IGMP [18], written more than 8 years after ICMP, with few additions to the parser lexicon. We also applied SAGE to the NTP RFC [51] and verified we could generate packets for the timeout procedure containing both NTP and UDP headers (which is NTP's default layer 3 protocol). Here too, SAGE needed minimal additions to the lexicon. Finally, SAGE's disambiguation is often very effective, reducing, in some sentences, 46 logical forms to 1.

Towards greater generality. SAGE is a first, but significant, step towards fully-automated processing of natural language protocol specifications and can already parse significant chunks of 3 standard RFCs. It is currently able to handle analysis, disambiguation, and code generation for the most common components including packet header diagrams, lists, pseudocode, and semantics of packet fields. More complex protocols contain state machine descriptions (*e.g.,* TCP) or specifications of how to process and update state (*e.g.,* BGP), which SAGE does not yet currently support. Other protocols contain software architecture descriptions (*e.g.,* OSPF, RTP) and communication patterns (*e.g.,* BGP 4). In §7, we break down the prevalence of these components by protocol to contextualize our contributions. Our sense is that extensions to handle protocol state and connection state will put SAGE within reach of being able to handle large parts of the TCP and BGP RFCs.

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

2 Background, Approach, and Overview

That protocol specifications can be ambiguous, and that these ambiguities lead to bugs and non-interoperability, is well known. In this section, we quantify this by examining the implementation of the Internet Control Message Protocol (ICMP [59]) by students in a graduate networking course. This scopes the specific problem we tackle in this paper.

2.1 Analysis 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.

To test whether students had implemented the echo reply message correctly, we used the Linux `ping` tool to send an echo message to their router (we tested their code using Mininet [42]). Across the 39 implementations, the Linux implementation correctly parsed the echo reply only for 24 of them (61.5%). The remaining 15 exhibited 6 categories (not mutually exclusive) of implementations errors (Table 1): mistakes in IP or ICMP header operations; byte order conversion errors; incorrectly generated ICMP payload in the echo reply message; an incorrect length for the payload; and a wrongly computed ICMP checksum (we discuss this in detail below). Each error category occurred in at least 4 of the 15 erroneous implementations.

To understand the incorrect checksum error 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 the header, or both the header and the payload. In fact, students came up with ***nine*** different interpretations (Table 2) including checksumming only the IP header, checksumming the ICMP header together with a few fixed extra bytes, and so on.

2.2 Approach and Overview

Dealing with Ambiguity. These results indicate that even a relatively simple RFC results in a large variety of interpretations and implementations. These ambiguities often result in non-interoperable implementations. RFC authors and the IETF community rely on manual methods to avoid or eliminate non-interoperabilities: careful review of standards drafts by participants, development of reference implementations, and interoperability ***bake-offs*** [31, 66] at which

INDEX	CODE REPRESENTATIONS
1	The size of destination unreachable message header.
2	The aggregate field size of type field, code field and checksum field
3	The received packet size deducts the size of ethernet and ip headers
4	The total length field in ip header deducts (internet header length field multiplies 4)
5	The size of ip header
6	The aggregate field size of type field, code field and checksum field plus 4
7	The total length field in ip header deducts size of ip header
8	Add the expected diff number to received packet's checksum field
9	Magic constants e.g. 2 or 8 or 36

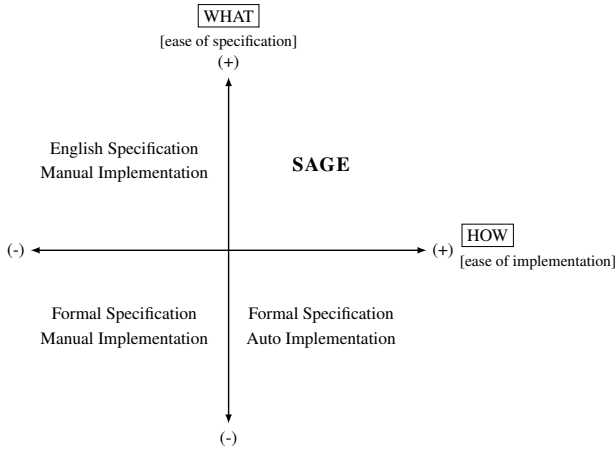**Table 2: Various code representations for checksum field computation length**

Figure 1: Approaches to specifying and implementing network protocols. SAGE aims to achieve the best of both worlds, with English specification and semi-automated implementation.

vendors and developers test their implementations against each other to discover non-interoperabilities 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 describe a single idea or concept; (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.

Structure in RFCs. On the flip side, unlike general English text, network protocol specifications have exploitable structure. The networking standards 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 [21] (something true of more recent RFCs) and all standards RFCs are carefully edited for clarity and adherence to style [63]. In recent years, *natural language processing* (NLP) has developed to the point where automated methods can exploit structure in stylized text to extract meaning.

Our approach. Motivated by these observations, in this paper, we take a first step towards *semi-automated protocol generation*, to synthesize functional code descriptions from natural language specifications, occupying a unique position in the design space (Figure 1). We leverage recent advances in the NLP problem of *semantic parsing* which seeks to extract semantics of a given piece of text. However, natural language is known [34, 62] to have lexical (*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.

For the foreseeable future we do not expect NLP to be able to unambiguously parse RFCs. Thus, we explore a *semi-automated* approach in which NLP tools, together with unit tests, help a human-in-the-loop iteratively discover and correct ambiguities. At the end of this process, the specification is amenable to automated code generation that interoperates with a reference implementation. We have instantiated this approach in a system called SAGE, which we discuss next.

SAGE. Figure 2 shows the three stages of SAGE. The *parsing* stage uses an NLP-based semantic parser [4] to generate intermediate representations, called *logical forms*, of RFC sentences. Because parsing isn't perfect, it can output multiple logical forms for the same sentence: each logical form corresponds to one semantic interpretation of the sentence, so multiple logical forms represent ambiguity. The *disambiguation* stage automatically, in some cases, eliminates such ambiguities. If, after this stage, ambiguities remain, SAGE asks a human to resolve them. The *code generator* compiles logical forms into executable code, a process that may also uncover specification ambiguity. The following sections describe these stages in greater detail.

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 [20, 39, 83] and certain types of automatic code generation [45, 60, 81]. However, such methods do not directly apply to our task. First, deep learning typically requires training in a "black-box" end-to-end manner. 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 are data-driven and require large-scale annotated data; 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 [4]) formalism that enables (a) coupling syntax and semantics in the parsing process and (b) is particularly suitable for handling domain-specific terminology by defining a small hand-crafted lexicon that encapsulates domain knowledge. CCG has successfully parsed natural language explanations into labeling rules in several contexts [69, 77].

CCG background. CCG takes as input a description of the syntax and semantics of natural language. 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
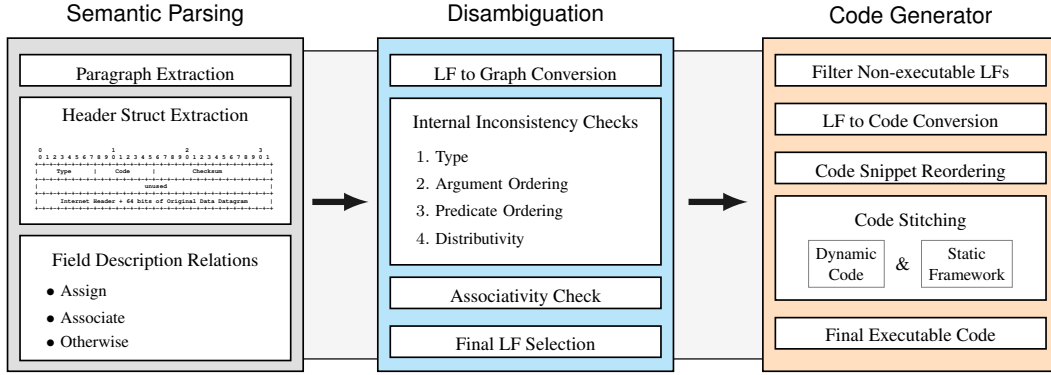
Figure 2: SAGE components

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

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

This expresses the fact (a) "checksum" is a special word in networking, that (b) "is" can represent assignment, and (c) zero can refer to a number.

CCG can use this lexicon to generate *logical form* that completely captures the semantics of a phrase such as "checksum is zero": {S: @IS"checksum", @NUM0}. Our code generator (§5) compiles these into executable code.

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). The following paragraphs describe how we address these challenges.

Specifying domain-specific syntax. The lexical entry (1) above specifies that checksum is a keyword in the vocabulary. Rather than using a human to 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, CCG, we also need to identify nouns and noun-phrases that occur generally in English, for which we use an NLP tool called SpaCy [28].

Specifying domain-specific semantics. CCG 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 other RFCs (see also section:evaluation). 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 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 several stylized elements which we extracted automatically using pre-processors we wrote. First, RFCs use indentation to represent content hierarchy and descriptive lists (*e.g.,* field names and their values). Our pre-processor extracts these relationships to aid in disambiguation (§4) and code generation (§5). Second, RFCs represent packet header fields (and associated field widths) using ASCII art; our pre-processor extracts field names and widths and directly generates data structures (specifically, structs in C) to represent headers to enable automated code generation (§5). Some RFCs [51] also contain pseudo-code; our pre-processor represents these as logical forms to facilitate code generation.

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

4 Disambiguation

In this section, we describe how SAGE can leverage domain knowledge to automatically resolve some ambiguities: cases where semantic parsing resulted in either 0 or more than 1 logical forms.

4.1 Why Ambiguities Arise

To motivate how we automatically resolve ambiguities, we take actual examples from the ICMP RFC. In all of these examples, 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. In all of these cases, the sentence is grammatically incomplete (specifically, it lacks 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

These kinds of sentences are common in protocol header field descriptions. The last sentence is convoluted and difficult 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

Specifically, 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 for reasons described below.

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 the action are clear from the semantics of 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 [27, 78].

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* [70][1]. 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, in some cases, CCG 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"[2]. 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. SAGE applies these checks to sentences with multiple

---

[1] For example: *Harry sees and Fred says he likes Sally.*

[2] 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.
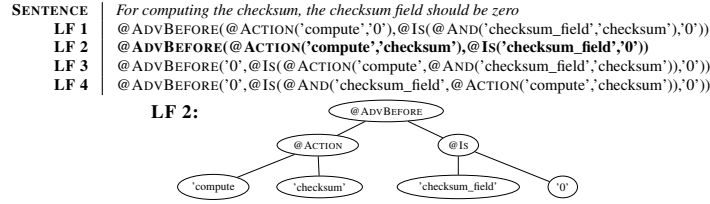
Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan

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'))

Figure 3: Example of multiple LFs from CCG parsing of "For computing the checksum, the checksum should be zero"
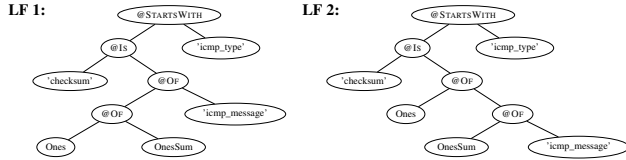
Figure 4: LF Graphs of *The checksum is the 16-bit one's complement of the one's complement sum of the ICMP message starting with the ICMP Type*

logical forms, thereby winnowing these usually down to one logical form (§6). These checks apply broadly because of the restricted way in which protocol specifications use natural language. While we have derived these by analyzing ICMP, we show that these checks also help disambiguate text in the IGMP [18] and NTP [51] 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 in which one predicate cannot be nested within another, SAGE defines checks that remove logical forms that violate the order.

Distributivity. To avoid incorrect semantics resulting from the 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* (identical modulo re-ordering of children of interior nodes). SAGE detects associativity using standard *graph isomorphism* detection algorithms.

5 Code Generation

Disambiguated logical forms are an intermediate representation ready to be converted to code, which we consider next.

5.1 Challenges

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

a protocol and the underlying system it runs on, (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 the one's complement function. 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 this assumed functionality along with an API to access and manipulate headers of other protocols, and to interface with the operating system. SAGE's generated code (discussed below) invokes the static framework. The framework may either contain a complete implementation of the protocols it abstracts, or, more likely, it invokes existing implementations of these protocols and existing services provided by the operating system.

Logical Forms as an Intermediate Representation. The semantic parser generates a logical form as a parsed representation of a sentence. From the perspective of 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 in the sentence "The checksum may be replaced in the future", or behavior intended for other protocols (*e.g.,* "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 must 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. In many cases, 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 in the RFC. 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 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

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

Table 3: Logical form with its context, and the resulting code snippet.

@ADVBEFORE), sentences that describe such *advices* (as used in functional and aspect-oriented languages[3]).

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 2. 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 (Table 3) a *context dictionary* for each logical form (or sentence) that provides context for code generation.

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 the 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, and concatenating these code snippets results in executable code for the logical form.

SAGE then concatenates code for all the logical forms in a message into a packet handling function. 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 applies to the sender or the receiver is also encoded in the context dictionary (Table 3). 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.

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

---

[3]Advices are statements associated with a function that must be executed before, after, or instead of that function. In our example, the checksum must be set to zero *before* computing the checksum.

TEST COMMANDS	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 10.0.1.1`	Test traceroute

Table 4: ICMP test commands used in project environment.

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. At least for the RFCs we have evaluated, SAGE can automatically tag such code generation failures as @ADVCOMMENT without human intervention (*i.e.,* there wasn't an instance of an actionable sentence that failed code generation once we had correctly defined the context).

6 Evaluation

In this section, we quantify SAGE's ability to discover specification ambiguities, the extent to which it generalizes across RFCs, and the importance of disambiguation and of our parsing and code generation extensions.

6.1 Methodology

Implementation. The key implementation additions needed for SAGE include a networking-specific dictionary, new CCG-parsable lexicon entries, a series of inconsistency checks, and logical form to code predicate handler functions. For the networking-specific dictionary, we used the index pages from [40] to build up a dictionary with about 400 networking terms. SAGE adds 71 lexical entries to an nltk-based CCG parser [47]. Overall, SAGE requires 7,128 lines of code.

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 therefore show the largest number among checks. The distributivity check has a single implicit rule. For code generation, we defined 25 predicate handler functions to convert LFs to code snippets.

Test Scenarios. Our primary evaluation focuses on the ICMP RFC, which defines eight ICMP message types including destination unreachable, time exceeded, parameter problem, source quench, redirect, echo/echo reply, timestamp/timestamp reply, and information request/reply. Like the student assignments we analyzed earlier, we generated code for each ICMP message type. We include the full, auto-generated implementation in §B for reference. To test generated code for each message, as with the student projects, we let the client be the sender and the router be the receiver. The client sends test messages to the router which then responds with the appropriate ICMP message. For each scenario, we dumped both sender and receiver packets with `tcpdump`. We verified all test scenarios using `tcpdump`. We also include details of each test scenario in the Appendix. To

demonstrate the generality of SAGE, we also evaluated IGMP [18] and NTP [51] (§6.3).

6.2 End-to-end Evaluation

Using two experiments, we verify whether ICMP code generated by SAGE produces packets that interoperate correctly with off-the-shelf tools.

Packet capture based verification. In the first experiment, we examined the packet emitted by a SAGE-generated ICMP implementation with `tcpdump` [71], to verify that `tcpdump` can read packet contents correctly without warnings or errors. Specifically, for each message type, for both sender and receiver cide, 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. In this experiment, we test whether a SAGE-generated ICMP receiver or router interoperates with mature probing tools like `ping` and `traceroute`. To do this, we integrated our static framework code and the SAGE-generated code into a Mininet-based framework used for the course projects described in §2. With this framework, we verified, with four Linux commands (Table 4), 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 four commands, the *generated code interoperates correctly with these probing tools*.

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 [18] and NTP [51]. 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 [18], 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 messages; we list the generated code in §C. 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 [51], 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 NTP. We list the generated code in §D.

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.
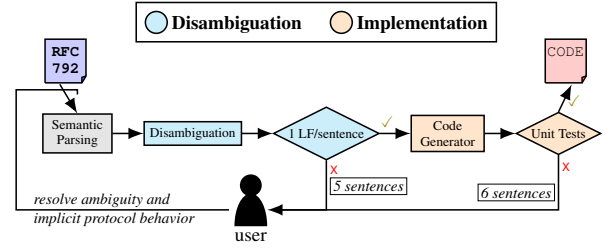

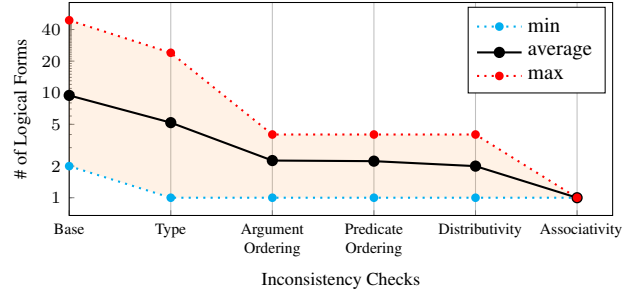**Figure 5: SAGE Workflow as of Processing RFC 792**

Figure 6: Number of Logical Forms after Inconsistency Checks measured on RFC 792 text: for each ambiguous sentence, sequentially executing the checks on logical forms (Base) reduces inconsistencies, and after the last Associativity check the final output is a single logical form.

6.4 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 spec authors or developers when it cannot, and can help iteratively verify re-written parts of the spec.

Ambiguous sentences. When we began to analyze RFC 792 with SAGE, we immediately encountered numerous ambiguities we have 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" is an example that 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
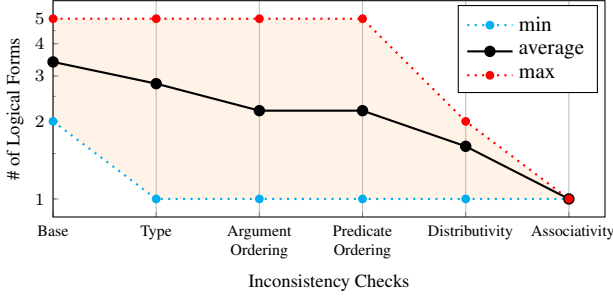
Figure 7: Number of Logical Forms after Inconsistency Checks measured on IGMPv1 field descriptions: for each ambiguous sentence, sequentially executing the checks on logical forms (Base) reduces inconsistencies, and after the last Associativity check the final output is a single logical form, as with ICMP.

Figure 8: Effectiveness of individual disambiguation checks on RFC 792 text: *Left:* **average number of logical forms filtered by the check per ambiguous sentence with standard error** *Right:* **number of ambiguous sentences affected of the total 42 sentences.**

SENTENCE	LABEL	LOGICAL FORMS
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 between good and poor noun phrase labels.

	MORE LFs	FEWER LFs	ZERO LF
Domain-specific Dictionary	17	0	0
Noun-phrase Labeling	0	8	54

Table 7: Disabling domain-specific dictionary and noun-phrase labeling

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 5. 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, 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, thereby reducing manual labor in disambiguation. To understand 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 6, 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 complete. 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 6 shows that all sentences resulted in 2-46 LFs, but SAGE's winnowing reduces this to 1 (after human-in-the-loop rewriting of
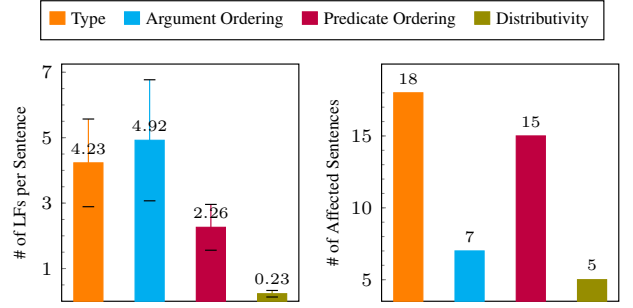
true ambiguities). Of these, type, argument ordering and the associativity checks are the most effective. We apply the same analysis to IGMP (Figure 7). In IGMP, the distributivity check is also important. This analysis shows the cumulative effect of applying checks in the order shown in the figure.

A more direct way to understand the efficacy of checks is shown in Figure 8 (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 discussed above, type and predicate ordering checks reduced LFs for the most number of sentences, but argument ordering reduced the most number of logical forms. For IGMP (omitted for brevity), in addition, the distributivity checks were also effective, reducing one LF every 2 sentences.

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 logical forms 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 one noun phrase, CCG outputs many more logical forms, making it harder to disambiguate the sentence. In the limit, when SAGE does

	RFC791 IPv4	RFC793 TCP	RFC768 UDP	RFC792 ICMP	RFC1059 NTP	RFC2328 OSPF v2	RFC4271 BGP 4	RFC3550 RTP
Packet Format	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
Communication Patterns	X	X				X	X	X
State/Session Management		X				X	X	X
Architecture					X	X		X

Table 8: Conceptual components in RFCs. SAGE supports the high-lighted components.

	RFC791 IPv4	RFC793 TCP	RFC768 UDP	RFC792 ICMP	RFC1059 NTP	RFC2328 OSPF v2	RFC4271 BGP 4	RFC3550 RTP
Packet Header Diagram	X	X	X	X	X	X	X	X
Listing	X	X	X	X	X	X	X	X
Table	X	X			X	X	X	X
Algorithm Description	X	X			X	X		X
Other Figures	X				X	X	X	X
Seq./Comm. Diagram	X	X			X	X		X
State Machine Diagram		X						

Table 9: Syntactic components in RFCs. SAGE supports the high-lighted components.

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: Peer variable sentence and the resulting code snippet.

not use careful noun phrase labeling, CCG is unable to parse some sentences at all (resulting in zero LF).

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 zero after winnowing.

7 SAGE Extensions

While SAGE takes a significant step towards parsing standards documents, much work (likely several papers worth!) remains.

Specification components. To understand this gap, we have manually inspected several protocol specifications and categorized components of specifications into two categories: syntactic and conceptual. 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. Most popular standards have many, if not all, of these elements. SAGE supports parsing of 3 of the 6 elements in the table, for ICMP and parts of NTP. Our results (§6.2) show that extending these elements to other protocols will require marginal extensions at each step. However, much work remains to achieve complete generality, of which state and session management is a significant piece.

SAGE is already able to parse state management. As an example, 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, as shown in Table 10, concerns when the timeout procedure is executed. SAGE is

able to parse the sentence into a LF and turn it into a code snippet. However, NTP requires (and other protocols may require) more complex co-reference resolution [26, 30]: in NTP, context for state management is distributed 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.

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.

This analysis suggests that, while much work remains, two significant protocols may be within reach: with the addition of state management and state machine diagrams, BGP. We plan to address these next.

Toward full automation? Ideally it would be possible to make translation of natural-language specifications to code a fully-automated process. Alas, we believe that the inherent complexities of free-form natural language combined with the inherent logical complexities of protocols and programming languages make it unlikely that this will ever be fully realizable. However, we believe that it is possible to come close to this, and have aimed to build SAGE as a first big step in this direction.

To this end, the key challenge would be to minimize the manual labor required to assist in disambiguation. Our winnowing already does this (§4.2), but future work will need to explore good user interfaces for human input when SAGE generates 0 LFs or more than 1 LF (Figure 5). SAGE will also need to develop ways for humans to specify cross-references (references to other protocols in a spec) and to write unit tests.

8 Related Work

Protocol Languages / Formal Specification Techniques. Numerous protocol languages have been proposed over the years. A few decades ago, Estelle [13] and LOTOS [11] sought to provide formal descriptions for OSI protocol suites. Although these formal description techniques can specify precise protocol behavior, it is hard for people to understand and thus use for specification or implementation. In addition, Estelle used finite state machine specs to depict how protocols communicate in parallel, passing on the complexity, unreadability, and rigidity of modification to followup work [12, 65, 75]. Other research such as RTAG [3], x-kernel [29], Morpheus [1], Prolac [38], and Network Packet Representation [50] gradually improved readability, structure, and performance of protocols, spanning specification 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 [9–11] developed techniques to reason about protocol behaviors in an effort to minimize bugs. Such techniques used specifications of finite state machines, higher-order

logic, or domain-specific languages to verify protocols. Another thread of work [36, 37, 43] explored the use of explicit-state model-checkers to find bugs in protocol implementations. This thread also inspired work (*e.g.,* [55]) 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 ambiguity in specifications 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 [79], rule-based analysis [24], statistic analysis [53, 74], and ML-based methods [67] to solve log analysis problems. Recent work [5, 8] has applied NLP-based techniques to extract semantic meanings from log files so that event categorization can be improved. SAGE is complementary to this line of work: it seeks to discover ambiguities in natural language specifications of networked systems.

Program Synthesis. To automatically generate code, prior work explored a range of techniques for program synthesis. Reactive synthesis [57, 58] 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.,* [68]) takes a correct-by-construction approach to develop inductive proofs to extract programs. Type-based synthesis [23, 54] takes advantage of the types provided from specification to refine output. In networking community, program synthesis techniques can automate (*e.g.,* [48, 49]) the updating of network configurations, and generating programmable switch code [25]. In the future, 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 natural language processing that aims to transform unstructured text into structured logical forms for subsequent execution [7]. 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 [17]. A SQL interpreter can execute this query on a database and give the correct answer "Toronto Argonauts" [33]. Apart from the application to question answering, semantic parsing has also been successful in navigating robots [72], understanding instructions [14], and playing language games [76]. Research in generating code from natural language requires one step beyond logical forms and aims to output concrete implementations in high-level general-purpose programming languages [46]. This problem is usually formulated as syntax-constrained sequence generation [44, 82]. The two topics are closely related to our work since the process of implementing network protocols following natural language specifications in RFCs requires the ability to understand and execute instructions.

Pre-trained Language Models. Recently, high-capacity pre-trained language models [19, 41, 56, 80] have greatly advanced machine performance in a variety of NLP tasks, including question answering, natural language inference, text classification, *etc.* The general practice is to first train a large model on huge corpus with unsupervised learning objectives (*i.e.,* pre-training); then re-use these weights to initialize a task-specific model which is later trained with labeled data (*i.e.,,* fine-tuning). In the context of SAGE, such pre-trained models advanced the state-of-the-art in semantic parsing [84, 85]. Recent work [22] also attempts to pre-train on programming language and natural language simultaneously, and achieves state-of-the-art performance in NL 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 [61], requires 8 M web pages for training.

9 Conclusions

This paper describes SAGE, which takes a first step towards semi-automated protocol code generation. SAGE includes domain-specific extensions to semantic parsing and automated discovery of ambiguities and automated disambiguation. It can generate code for ICMP that interoperates correctly with Linux implementations of `ping` and `traceroute`. SAGE is able to discover previously-missed ambiguities. Future work can extend SAGE to parse more protocol components, and devise better methods to involve humans in the loop to detect and fix ambiguities and guide the search for bugs.

References

[1] ABBOTT, M. B., AND PETERSON, L. L. A language-based approach to protocol implementation. *IEEE/ACM transactions on networking 1*, 1 (1993), 4–19.

[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., FITZGERALD, N., AND ZETTLEMOYER, L. S. Semantic parsing with combinatory categorial grammars. *ACL (Tutorial Abstracts) 3* (2013).

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] BOLOGNESI, T., AND BRINKSMA, E. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems 14*, 1 (1987), 25–59.

[12] BOUSSINOT, F., AND DE SIMONE, R. The esterel language. *Proceedings of the IEEE 79*, 9 (1991), 1293–1304.

[13] 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.

[14] 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).

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

[16] D. HARKINS, E. Secure Password Ciphersuites for Transport Layer Security

(TLS). RFC 8492, 2019.

[17] 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.

[18] DEERING, D. S. E. Host extensions for IP multicasting. RFC 1112, 1989.

[19] 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).

[20] DONG, L., AND LAPATA, M. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793* (2018).

[21] EDITOR, R., AND FLANAGAN, H. RFC Style Guide. RFC 7322, Sept. 2014.

[22] 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).

[23] FESER, J. K., CHAUDHURI, S., AND DILLIG, I. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices 50*, 6 (2015), 229–239.

[24] 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.

[25] 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.

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

[27] 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.

[28] HONNIBAL, M., AND MONTANI, I. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[29] 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.

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

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

[32] JETHANANDANI, M., AGARWAL, S., HUANG, L., AND BLAIR, D. YANG Data Model for Network Access Control Lists (ACLs). RFC 8519, 2019.

[33] KAMATH, A., AND DAS, R. A survey on semantic parsing. *arXiv preprint arXiv:1812.00978* (2018).

[34] KEMPSON, R. M., AND CORMACK, A. Ambiguity and quantification. *Linguistics and Philosophy 4*, 2 (1981), 259–309.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] KUROSE, J., AND ROSS, K. Computer networking: A top down approach, 2012.

[41] 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).

[42] 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.

[43] LEE, H., SEIBERT, J., KILLIAN, C. E., AND NITA-ROTARU, C. Gatling: Automatic attack discovery in large-scale distributed systems. In *NDSS* (2012), Citeseer.

[44] 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).

[45] 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).

[46] LING, W., GREFENSTETTE, E., HERMANN, K. M., KOČISKÝ, T., SENIOR, A., WANG, F., AND BLUNSOM, P. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[47] LOPER, E., AND BIRD, S. Nltk: the natural language toolkit. *arXiv preprint cs/0205028* (2002).

[48] MCCLURG, J., HOJJAT, H., ČERNÝ, P., AND FOSTER, N. Efficient synthesis of network updates. *Acm Sigplan Notices 50*, 6 (2015), 196–207.

[49] MCCLURG, J., HOJJAT, H., FOSTER, N., AND ČERNÝ, P. Event-driven network programming. *ACM SIGPLAN Notices 51*, 6 (2016), 369–385.

[50] 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.

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

[52] 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.

[53] 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.

[54] OSERA, P.-M., AND ZDANCEWIC, S. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices 50*, 6 (2015), 619–630.

[55] 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.

[56] 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).

[57] 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.

[58] 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.

[59] POSTEL, J. Internet Control Message Protocol. RFC 792, 1981.

[60] RABINOVICH, M., STERN, M., AND KLEIN, D. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[61] 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.

[62] 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.

[63] Rfc editor. http://www.rfc-editor.org/.

[64] S. GUERON, A. LANGLEY, Y. L. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. RFC 8452, 2019.

[65] SIDHU, D., AND CHUNG, A. *A formal description technique for protocol engineering.* University of Maryland at College Park, 1990.

[66] First sip interoperability test event. https://www.cs.columbia.edu/sip/sipit/1/, 2008.

[67] 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.

[68] 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.

[69] 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.

[70] STEEDMAN, M., AND BALDRIDGE, J. Combinatory categorial grammar. *Non-Transformational Syntax: Formal and explicit models of grammar* (2011), 181–224.

[71] Tcpdump & libpcap public repository. https://www.tcpdump.org/. Accessed: 2020-05-22.

[72] 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).

[73] THOMSON, M. Example Handshake Traces for TLS 1.3. RFC 8448, 2019.

[74] 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.

[75] 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.

[76] WANG, S. I., LIANG, P., AND MANNING, C. D. Learning language games through interaction. *arXiv preprint arXiv:1606.02447* (2016).

[77] 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).

[78] 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.

[79] 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.

[80] 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.

[81] YIN, P., AND NEUBIG, G. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[82] 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, pp. 440–450.

[83] 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).

[84] ZHANG, S., MA, X., DUH, K., AND DURME, B. V. Amr parsing as sequence-to-graph transduction. *ArXiv abs/1905.08704* (2019).

[85] ZHANG, S., MA, X., DUH, K., AND DURME, B. V. Broad-coverage semantic parsing as transduction. In *EMNLP/IJCNLP* (2019).

Appendix

A   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 call the generated function to construct destination unreachable message back to the sender.

Time Exceeded Message. At the sender side, we intentionally generate a packet that the time-to-live field in IP header is one, and the destination IP address is set to the server1 address. At the router side, the router has a check to the packet header 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 type of service value to one. The router interface recognizes the unsupported type of service value and call 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 new datagram. At the sender side, we generated a packet to the 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 send to the router. The router discovered the next gateway is in the same subnet as the sender host, and therefore construct the redirect message to the client with the redirect gateway address by calling the generated functions.

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

Timestamp and Timestamp Reply Message. The sender and receiver behavior of this scenario is identical to echo and echo reply message one. The sender sends packet by calling generated function and the receiver matches the ICMP type and replies packets with the generated function. The difference lies in the packet generated by 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 separate 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 messages. Similar to the timestamp/timestamp reply message, the differences lie in the generated packets which do not have datagram data and the field values are different.

B The Generated ICMP Code

This sections presents the C/C++ code generated by SAGE for ICMP protocol as it was described in RFC 792 [59]. The generated code relies on a static framework. For brevity, we present only the helper functions used in generated code in §B.1. We present two types of generated codes: message headers generated from ASCII-art figures (§B.3) and packet handling functions from field descriptions (§B.2). All generated codes are produced by SAGE without further edits.

B.1 Helper Functions The generated code relies on knowledge outside of the target RFC. This knowledge is implemented by helper functions and data structures. Declarations of these functions and structs are presented in Listing 1. These functions are part of the static framework, hence none of them are generated.

```
// pointers to each head of protocol
struct proto_ptr {
  uint8_t* eth_ptr;
  uint8_t* ip_ptr;
  uint8_t* icmp_ptr;
};
typedef struct proto_ptr proto_ptr_t;

// Returns true if the argument 'x' is an odd number
bool isodd(uint32_t x);

// Pad 'data_len' size of 'data' with 'pad_len' size of 'pad_char'
void pad(char **data, int shift_byte, int data_len, int pad_len, char pad_char);

// Wrapper for memcpy()
void copy(char **buffer, char* data, int len);

// Return 16-bit one's complement of the argument 'number'
uint16_t u16bit_ones_complement(uint32_t x);

// Return one's complement sum of the argument 'data' for 'len' size
uint32_t ones_complement_sum(const void *data, int len);

// Return the redirect gateway addr for the dest address from ip
// header pointed by argument 'ip_ptr'
uint32_t find_gateway(uint8_t* ip_ptr);

// Return the group addr for the dest address from ip header pointed
// by argument 'ip_ptr'
uint32_t find_group_addr(uint8_t* ip_ptr);

// Do nothing
void dummy_action();
```

Listing 1: Static Helper Functions

B.2 Packet Handling Functions SAGE-generated packet handling functions for ICMP messages are presented in Listing 2. Beside the include directives, this file is generated. Comments are taken from input texts. Each function follows a `fill_icmp_{message_type}_{role}` naming scheme. For ICMP, two roles are defined; sender if it creates a request message such as Echo Message, and receiver if it creates a reply message such as Echo Reply Message. If there is no difference in sender and receiver codes, only the receiver function is generated. This is common among ICMP error messages.

```c
#ifndef GEN_H_
#define GEN_H_

#include "helper.h"
#include "icmp_hdr.h"
#include "meta.h"
#include "proto.h"

void fill_icmp_dest_unreachable_receiver(
    Destination_Unreachable_Message_hdr *hdr, uint16_t length, int code_value,
    proto_ptr_t *ptrs) {
  char *payload = (char *) (hdr + 1);
  // The source network and address from the original datagram's data
  copy(&payload, (char *) ptrs->ip_ptr, 28);
  // Set type to 3
  hdr->type = 3;
  // 0 = net unreachable;
  // 1 = host unreachable;
  // 2 = protocol unreachable;
  // 3 = port unreachable;
  // 4 = fragmentation needed and DF set;
  // 5 = source route failed
  hdr->code = code_value;
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
      ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_echo_receiver(Echo_or_Echo_Reply_Message_hdr *hdr,
                             uint16_t length, int type_value) {
  char *data = (char *) (hdr + 1);
  // 0 for echo reply message
  hdr->type = type_value;
  // Set code to 0
  hdr->code = 0;
  // If code equals 0, an identifier may be zero to help match echos and replies
  if (hdr->code == 0) {
    hdr->identifier = 0;
  }
  // If code equals 0, a sequence number may be zero to help match echos and
  // replies
  if (hdr->code == 0) {
    hdr->sequence_number = 0;
  }
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // For computing the checksum, if the total length is odd, the received data
  // is padded with one octet of zeros
  if (isodd(length)) {
    pad(&data, sizeof(*data), 0, 1);
    length += 1;
  }
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
      ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_echo_sender(Echo_or_Echo_Reply_Message_hdr *hdr, uint16_t length,
                           int type_value) {
  char *data = (char *) (hdr + 1);
  // 8 for echo message;
  // 0 for echo reply message
  hdr->type = type_value;
  // Set code to 0
  hdr->code = 0;
  // If code equals 0, an identifier may be zero to help match echos and replies
  if (hdr->code == 0) {
    hdr->identifier = 0;
  }
  // If code equals 0, a sequence number may be zero to help match echos and
  // replies
  if (hdr->code == 0) {
    hdr->sequence_number = 0;
  }
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // For computing the checksum, if the total length is odd, the received data
  // is padded with one octet of zeros
  if (isodd(length)) {
    pad(&data, sizeof(*data), 0, 1);
    length += 1;
  }
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
```

```c
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_info_receiver(
    Information_Request_or_Information_Reply_Message_hdr *hdr, uint16_t length,
    int type_value) {
  // The type field changed to 16
  type_field = 16;
  // 15 for information request message;
  // 16 for information reply message
  hdr->type = type_value;
  // Set code to 0
  hdr->code = 0;
  // If code equals 0, an identifier may be zero to help match request and
  // replies
  if (hdr->code == 0) {
    hdr->identifier = 0;
  }
  // If code equals 0, a sequence number may be zero to help match request and
  // replies
  if (hdr->code == 0) {
    hdr->sequence_number = 0;
  }
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_para_prob_receiver(Parameter_Problem_Message_hdr *hdr,
                                  uint16_t length, proto_ptr_t *ptrs) {
  char *payload = (char *) (hdr + 1);
  // The source network and address from the original datagram's data
  copy(&payload, (char *) ptrs->ip_ptr, 28);
  // Set type to 12
  hdr->type = 12;
  // 0 = pointer indicates the error
  hdr->code = 0;
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_redir_receiver(Redirect_Message_hdr *hdr, uint16_t length,
                              int code_value, proto_ptr_t *ptrs) {
  char *payload = (char *) (hdr + 1);
  // The source network and address of the original datagram's data
  copy(&payload, (char *) ptrs->ip_ptr, 28);
  // Set type to 5
  hdr->type = 5;
  // 0 = Redirect datagrams for the Network
  // 1 = Redirect datagrams for the Host
  // 2 = Redirect datagrams for the Type of Service and Network
  // 3 = Redirect datagrams for the Type of Service and Host
  hdr->code = code_value;
  // Gateway Internet Adrress is the address of the gateway of the internet
  // destination address of the original datagram's
  hdr->gateway_internet_address = find_gateway(ptrs->ip_ptr);
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_source_quench_receiver(Source_Quench_Message_hdr *hdr,
                                      uint16_t length, proto_ptr_t *ptrs) {
  char *payload = (char *) (hdr + 1);
  // The source network and address of the original datagram's data
  copy(&payload, (char *) ptrs->ip_ptr, 28);
  // Set type to 4
  hdr->type = 4;
  // Set code to 0
  hdr->code = 0;
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_time_exceed_receiver(Time_Exceeded_Message_hdr *hdr,
                                    uint16_t length, int code_value,
                                    proto_ptr_t *ptrs) {
  char *payload = (char *) (hdr + 1);
  // The source network and address from the original datagram's data
  copy(&payload, (char *) ptrs->ip_ptr, 28);
  // Set type to 11
  hdr->type = 11;
  // 0 = time to live exceeded in transit;
  // 1 = fragment reassembly time exceeded
  hdr->code = code_value;
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_timestamp_receiver(Timestamp_or_Timestamp_Reply_Message_hdr *hdr,
                                  uint16_t length, int type_value) {
  // The type field changed to 14
  type_field = 14;
  // 14 for timestamp reply message
  hdr->type = type_value;
  // Set code to 0
  hdr->code = 0;
  // If code equals 0, an identifier may be zero to help match timestamp and
  // replies
  if (hdr->code == 0) {
    hdr->identifier = 0;
  }
  // If code equals 0, a sequence number may be zero to help match timestamp and
  // replies
  if (hdr->code == 0) {
    hdr->sequence_number = 0;
  }
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

void fill_icmp_timestamp_sender(Timestamp_or_Timestamp_Reply_Message_hdr *hdr,
                                uint16_t length, int type_value) {
  // The type field changed to 14
  type_field = 14;
  // 13 for timestamp message;
  // 14 for timestamp reply message
  hdr->type = type_value;
  // Set code to 0
  hdr->code = 0;
  // If code equals 0, an identifier may be zero to help match timestamp and
  // replies
  if (hdr->code == 0) {
    hdr->identifier = 0;
  }
  // If code equals 0, a sequence number may be zero to help match timestamp and
  // replies
  if (hdr->code == 0) {
    hdr->sequence_number = 0;
  }
  // For computing the checksum , the checksum field should be zero
  hdr->checksum = 0;
  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the ICMP message starting with the ICMP Type
  hdr->checksum = u16bit_ones_complement(
        ones_complement_sum((const void *) &hdr->type, length));
}

#endif  // GEN_H_
```

Listing 2: The Generated ICMP Packet Handling Functions

B.3 Message Headers Message headers are generated by parsing RFC 792 ASCII-art figures, and are shown in Listing 3. Each message type pair (ie, Echo and Echo Reply) is described by a struct object. Fields with known sizes are represented as appropriate-length integer types or bitfields. If the size can not be guessed, comments are added to advise representation.

```c
#ifndef ICMP_HDR_H_
#define ICMP_HDR_H_

struct Destination_Unreachable_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint32_t unused;
  // char * internet_header_w_64_bits_of_original_data_datagram: 224;
};

struct Time_Exceeded_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint32_t unused;
  // char * internet_header_w_64_bits_of_original_data_datagram: 224;
};

struct Parameter_Problem_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint8_t pointer;
  uint64_t unused: 24;
  // char * internet_header_w_64_bits_of_original_data_datagram: 224;
};

struct Source_Quench_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint32_t unused;
  // char * internet_header_w_64_bits_of_original_data_datagram: 224;
};

struct Redirect_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint32_t gateway_internet_address;
  // char * internet_header_w_64_bits_of_original_data_datagram: 224;
};

struct Echo_or_Echo_Reply_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint16_t identifier;
  uint16_t sequence_number;
  // char * data;
};

struct Timestamp_or_Timestamp_Reply_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint16_t identifier;
  uint16_t sequence_number;
  uint32_t originate_timestamp;
  uint32_t receive_timestamp;
  uint32_t transmit_timestamp;
};

struct Information_Request_or_Information_Reply_Message_hdr {
  uint8_t type;
  uint8_t code;
  uint16_t checksum;
  uint16_t identifier;
  uint16_t sequence_number;
};

#endif  // ICMP_HDR_H_
```

Listing 3: ICMP Message Headers Generated From ASCII-art

C The Generated IGMP Code

This section presents the packet handling functions and message headers generated during the verification of SAGE's generality (discussed in §6.3).

C.1 Packet Handling Functions Packet handling functions for IGMP messages generated from the appendix of RFC 1112 [18] are presented in Listing 4.

```c
#ifndef IGMP_GEN_H_
#define IGMP_GEN_H_

void fill_igmp_igmp_query(INTERNET_GROUP_MANAGEMENT_PROTOCOL_hdr *hdr,
                          uint16_t length, int type_value, proto_ptr_t *ptrs) {
  // Set version to 1
  hdr->version = 1;

  // Unused field is zeroed when sent
  /*send*/
  hdr->unused = 0;

  // Unused field is ignored when received.
  // The group address field is ignored when received.
  /*receive*/
  dummy_action();

  // For a Host Membership Query message, the group address field is zeroed when
  // sent
  /*send*/
  hdr->group_address = 0;

  // 1 = Host Membership Query,
  // 2 = Host Membership Report.
  hdr->type = type_value;

  // For computing the checksum, the checksum field is zero.
  hdr->checksum = 0;

  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the IGMP message
  hdr->checksum =
      u16bit_ones_complement(ones_complement_sum((const void *) hdr, length));
}

void fill_igmp_igmp_report(INTERNET_GROUP_MANAGEMENT_PROTOCOL_hdr *hdr,
                           uint16_t length, int type_value, proto_ptr_t *ptrs) {
  // Set version to 1
  hdr->version = 1;

  // Unused field is zeroed when sent
  /*send*/
  hdr->unused = 0;

  // Unused field is ignored when received.
  // The group address field is ignored when received.
  /*receive*/
  dummy_action();

  // 1 = Host Membership Query,
  // 2 = Host Membership Report.
  hdr->type = type_value;

  // For a Host Membership Report message, the group address field holds the IP
  // host group address of the group being reported.
  hdr->group_address = find_group_addr(ptrs->ip_ptr);

  // For computing the checksum, the checksum field is zero.
  hdr->checksum = 0;

  // The checksum is the 16-bit one's complement of the one's complement sum of
  // the IGMP message
  hdr->checksum =
    u16bit_ones_complement(ones_complement_sum((const void *) hdr, length));
}

#endif  // IGMP_GEN_H_
```

Listing 4: The Generated IGMP Packet Handling Functions

C.2 Message Headers The message header in Listing 5 is generated from the appendix of RFC 1112 [18].

```c
#ifndef IGMP_HDR_H_
#define IGMP_HDR_H_

struct INTERNET_GROUP_MANAGEMENT_PROTOCOL_hdr {
  uint64_t version: 4;
  uint64_t type: 4;
  uint8_t unused;
  uint16_t checksum;
  uint32_t group_address;
};

#endif  // IGMP_HDR_H_
```

Listing 5: IGMP Message Headers generated from ASCII-art

D The Generated NTP Code

This section presents the packet handling functions and message headers generated during the verification of SAGE's generality.

D.1 Packet Handling Functions Packet handling functions for NTP data format are generated from the appendix of RFC 1059 [51] are presented in Listing 6.

```c
#ifndef NTP_GEN_H_
#define NTP_GEN_H_

void fill_ntp_ntp_data_format_timeout(NTP_Data_Format_hdr *hdr,
                                      uint16_t length) {
  // Leap Indicator is a Two-bit code warning of impending leap-second
  // Leap Indicator is inserted at the end of the last day of the current month
  // Bits are coded as follows:
  // 00 = no warning day has 86400 seconds
  // 01 = +1 second day has 86401 seconds
  // 10 = -1 second day has 86399 seconds
  // 11 = alarm condition clock not synchronized
  // pkt.leap <- sys.leap
  hdr->li = sys.leap;

  // Version Number is a Three-bit code indicating the version number
  // pkt.version <- NTP.VERSION
  hdr->vn = NTP.VERSION;

  // Stratum is an Integer identifying stratum level of local clock
  // Values are defined as follows:
  // 0 = unspecified
  // 1 = primary reference synchronized by outside means
  // 2...n = secondary reference via NTP
  // pkt.stratum <- sys.stratum
  hdr->stratum = sys.stratum;

  // Poll is a Signed integer indicating the maximum interval between successive
  // messages Poll is in seconds to the nearest power of two
  // pkt.poll <- peer.hpoll
  hdr->poll = peer.hpoll;

  // Precision is a Signed integer indicating the precision of the local clock
  // Precision is in seconds to the nearest power of two.
  // pkt.precision <- sys.precision
  hdr->precision = sys.precision;

  // synchronizing distance is a Fixed-point number indicating the estimated
  // roundtrip delay to the primary synchronizing source
  // pkt.distance <- sys.distance
  hdr->synchronizing_distance = sys.distance;

  // Estimated drift rate is a Fixed-point number indicating the estimated drift
  // rate of the local clock
  // pkt.drift <- sys.drift
  hdr->estimated_drift_rate = sys.drift;

  // reference clock identifier is a code identifying the particular reference
  // clock
  // pkt.refid <- sys.refid
  hdr->reference_clock_identifier = sys.refid;

  // reference timestamp is the Local time at which the local clock was last
  // set or corrected.
  // pkt.reftime <- sys.reftime
  hdr->reference_timestamp_64_bits = sys.reftime;

  // originate timestamp is the Local time at which the request departed the
  // client host for the service host
```

```c
  // pkt.org <- sys.clock
  hdr->originate_timestamp_64_bits = sys.clock;

  // receive timestamp is the Local time at which the request arrived at the
  // service host.
  // pkt.rec <- sys.clock
  hdr->receive_timestamp_64_bits = sys.clock;

  // transmit timestamp is the Local time at which the reply departed the
  // service host for the client host.
  // pkt.xmt <- sys.clock
  hdr->transmit_timestamp_64_bits = sys.clock;
}

#endif  // NTP_GEN_H_
```

Listing 6: The Generated NTP Packet Handling Functions

```c
#ifndef NTP_UDP_GEN_H_
#define NTP_UDP_GEN_H_

void fill_ntp_udp_receiver(UDP_Header_Format_hdr *hdr, uint16_t length) {
  // Length is length of the NTP message adding length of UDP header
  hdr->length = compute_length(struct NTP_Data_Format_hdr) +
                compute_length(struct UDP_Header_Format_hdr);

  // In the case of a client request, this field is assigned port 123
  if (client_request) {
    hdr->destination_port = 123;
  }

  // In the case of symmetric mode, Source Port and Destination Port fields are
  // assigned port 123
  if (symmetric_mode) {
    hdr->destination_port = 123;
  }

  // In the case of symmetric mode, Source Port and Destination Port fields are
  // assigned port 123
  if (symmetric_mode) {
    hdr->source_port = 123;
  }

  // Compute the standard UDP checksum.
  compute_standard_udp_checksum();
}

#endif  // NTP_UDP_GEN_H_
```

Listing 7: The Generated NTP-UDP Packet Handling Functions

The NTP messages are encapsulated in UDP packets. The field values of the UDP packets are described in the RFC. We present the code generated from the RFC description in Listing 7.

D.2 Message Headers The NTP message header (Listing 8) and the UDP packet header (Listing 9) are generated from the appendix of RFC 1059 [51].

```c
#ifndef NTP_HDR_H_
#define NTP_HDR_H_

struct NTP_Data_Format_hdr {
  uint64_t li: 2;
  uint64_t vn: 3;
  uint64_t reser: 3;
  uint8_t stratum;
  uint8_t poll;
  uint8_t precision;
  uint32_t synchronizing_distance;
  uint32_t estimated_drift_rate;
  uint32_t reference_clock_identifier;
  uint64_t reference_timestamp_64_bits;
  uint64_t originate_timestamp_64_bits;
  uint64_t receive_timestamp_64_bits;
  uint64_t transmit_timestamp_64_bits;
};

#endif  // NTP_HDR_H_
```

Listing 8: NTP Data Format Packet Headers Generated from ASCII-art

For $For	computing $Compute	the $The	checksum "checksum"	, $Punctuate	the $The	checksum "checksum"	should $ModalVerb	be $Is	zero. "0"
(S/S)/NP	*NP/NP*	*NP/NP*	*NP*	*S/S*	*NP/NP*	*NP*	*(S\NP)/(S\NP)*	*(S\NP)/NP*	*NP*
λx.λy.@AdvBefore(x,y)	λx.@Action(compute, x)	λx.x	"checksum"	λx.x	λx.x	"checksum"	λx.x	λy.λx.@Is(x, y)	"0"

NP "checksum"   ·   *NP* "checksum"   ·   *S\NP* λx.@Is(x, "0")

NP @Action("compute", "checksum")   ·   S\NP λx.@Is(x, "0")

S/S λy.@AdvBefore(@Action("compute", "checksum"), y)   ·   *S* @Is("checksum", "0")

S @Is("checksum", "0")

S @AdvBefore(@Action("compute", "checksum"), @Is("checksum", "0"))

Figure 9: Constructing one logical form from the sentence "For computing the checksum, the checksum should be zero" with CCG.

```
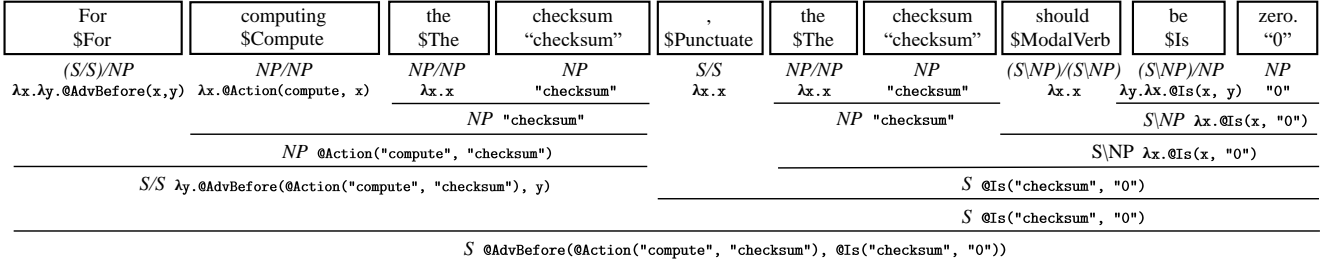#ifndef NTP_UDP_HDR_H_
#define NTP_UDP_HDR_H_

struct UDP_Header_Format_hdr {
  uint16_t source_port;
  uint16_t destination_port;
  uint16_t length;
  uint16_t checksum;
};

#endif  // NTP_UDP_HDR_H_
```

Listing 9: UDP Packet Headers for NTP Messages Generated from ASCII-art

E Complex CCG Parsing Example

We show a complex example of deriving one final logical form from the sentence: "For computing the checksum, the checksum should be zero." in Figure 9. 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, CCG parsing algorithm automatically applies combination rules and derives final logical forms for the whole sentence.