University of Glasgow | School of Computing Science

Parsing State Machine Models from Standards Documents

Mairi Sillars Moya (2404079)

April 19, 2023

ABSTRACT

The standards that build the Internet are written in natural language as opposed to more structured and formal methods. Whilst natural language can be useful, its use causes ambiguous and imprecise standards to be written, leading to errors in implementations. In this work, we propose the introduction of usable formal methods into the definition of state machines in standards documents. This formality enables automated parsing, which can be used to generate partial implementations to aid in reducing bugs and errors.

1. INTRODUCTION

The protocol standards that define the Internet are written in English prose. This use of natural language is beneficial because it allows for wide participation in the standards process, but it also has some drawbacks. It can be imprecise and ambiguous, and this can lead to mistakes and security vulnerabilities in implementations. The use of structured specification languages and formal methods has been proposed to reduce these ambiguities and improve the quality of specifications.

Writing full RFCs in a formal language is not feasible. They are lengthy documents and are written to be read by humans, for whom it is easier to read natural language. However, there are certain aspects of RFCs which lend themselves to formalisation more than others. Some previous work in the field that investigates this type of formalisation. McQuistin et al. [7] propose a type system for the description of the format, parsing and serialisation of protocols, which allows for flexibility in the syntax used for describing packets, whilst simultaneously providing sufficient formality to allow for automated parsing. Further to this work, McQuistin et al. [8] investigate code generation for parsing network packets using the type system developed in [7], and find that a formal syntax similar to that already present in RFCs can be parsed to generate code, showing that a balance between formal methods and usability is possible.

In this paper, we consider how state machines are represented in network protocol standards documents, and whether it is possible to represent these in a format which is generalisable to most network protocols. Specifically, we present a formal syntax which is similar to the method used in the TCP RFC [1] for textually defining state machines, a structured intermediate representation of state machines which is machine readable and allows for flexibility in the syntax used by authors, as well as a method to generate partial code implementations, which provide a base structure of protocols for the later development of full implementations.

Most of the work that deals with ambiguity in RFCs addresses other areas of them, such as packet parsing [7], constraints and field descriptions [18]. The work that addresses state machine descriptions in RFCs [9] does so from a natural language processing perspective. In this work we propose addressing issues that arise from ambiguities present in RFCs through the parsing of state machines, achieved by defining a structured but flexible formal definition of these.

We structure the remainder of this paper as follows. In Section 2 we discuss the representations used in current RFCs for the description of state machines, and the type of state machines which should be used for the representation of network protocols. In Section 3 we discuss inconsistencies in these descriptions, and the issues this poses to automated parsing of RFCs. In Section 4 we motivate and propose a standardised intermediate representation of state machines for network protocols. In Section 5 we discuss a syntax designed to parse state machines from RFCs. In Section 6 we discuss the benefits of code generation enabled by the intermediate representation of state machines. In Section 7 we critically evaluate the intermediate representation, syntax and code generation presented in the project. In Section 8 we discuss the related work in this field. Finally, in Section 9 we conclude the work.

2. REPRESENTING STATE MACHINES

As we have discussed previously, many RFCs contain some form of state machine description. However, the representation of these state machines varies across RFCs, and there is no formal structure by which they are written. In this section we discuss the styles currently used for the description of state machines, as well as the specific types of state machines that should be used to represent network protocols.

State Machines in RFCs

Several RFCs contain ASCII art diagrams, where states and transitions are displayed visually. An example of this style of representation is the state machine provided for a TLS 1.3 client, shown in Figure 1.

Other RFCs contain textual descriptions of state machines, where the states and transitions are discussed in plain text. These textual descriptions can be varied in the level of detail they provide. An example of a high-level textual description of a state machine is the basic operation description found in the POP3 RFC, shown in Figure 2.

As well as this, some RFCs provide a combination of both these methods. TCP [1] contains both an ASCII diagram as a summary overview of its state machine, as well as a detailed event processing section which describes state transitions textually. Finally, some protocols, such as DCCP [5], contain pseudocode descriptions of their state machines, as

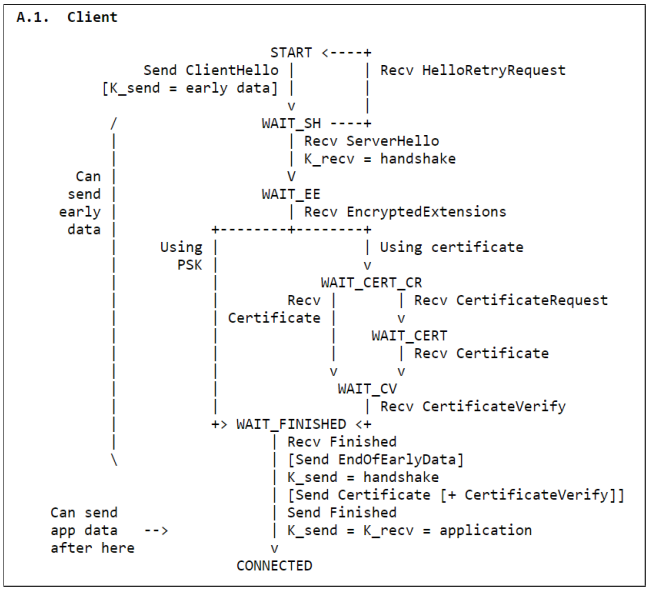


Figure 1: ASCII diagram representing TLS 1.3 client, RFC 8446 [14]

A POP3 session progresses through a number of states during its lifetime. Once the TCP connection has been opened and the POP3 server has sent the greeting, the session enters the AUTHORIZATION state. In this state, the client must identify itself to the POP3 server. Once the client has successfully done this, the server acquires resources associated with the client's maildrop, and the session enters the TRANSACTION state. In this state, the client requests actions on the part of the POP3 server. When the client has issued the QUIT command, the session enters the UPDATE state. In this state, the POP3 server releases any resources acquired during the TRANSACTION state and says goodbye. The TCP connection is then closed.

Figure 2: POP3 state machine overview, RFC 1939 [15]

shown in Figure 3.

Whilst there are several methods used to describe state machines, we argue that the differences between them are mainly syntactical. The representations refer to the same concept, implying that it is possible to express the underlying state machines using any of these methods interchangeably. However, different methods have varying degrees of expressiveness. For example, ASCII diagrams are often not powerful enough to describe a full state machine. The TCP RFC [1] notes that the state machine diagram provided is a summary, and should not be taken as its full specification. Similarly, the state machines provided in the TLS 1.3 RFC [14] are described as summaries. Whilst no explicit transitions are described throughout the document, the functionality of the protocol is also expressed textually, and can be used in combination with the diagram provided to extract a fuller representation of its state machines.

Textual descriptions also vary in terms of expressiveness. Some are more structured, thorough and explicit than others. For example, TCP provides a list of possible states and events that take place throughout the protocol followed by detailed event processing descriptions defining the effect of events on each possible state in the machine. POP3, on the other hand, does not list all possible states or events, and is not as explicit or systematic in describing the handling

```
Step 1: Check header basics
   /* This step checks for malformed packets.  Packets that fail
      these checks are ignored -- they do not receive Resets in
      response */
   If the packet is shorter than 12 bytes, drop packet and return
   If P.type is not understood, drop packet and return
   If P.Data Offset is smaller than the given packet type's
         fixed header length or larger than the packet's length,
         drop packet and return
   If P.type is not Data, Ack, or DataAck and P.X == 0 (the packet
         has short sequence numbers), drop packet and return
   If the header checksum is incorrect, drop packet and return
   If P.CsCov is too large for the packet size, drop packet and
         return

Step 2: Check ports and process TIMEWAIT state
   /* Flow ID is <src addr, src port, dst addr, dst port> 4-tuple */
   Look up flow ID in table and get corresponding socket
   If no socket, or S.state == TIMEWAIT,
      /* The following Reset's Sequence and Acknowledgement Numbers
         are taken from the input packet; see Section 8.3.1. */
      Generate Reset(No Connection) unless P.type == Reset
      Drop packet and return
```

Figure 3: Exctract of DCCP pseudocode provided in RFC 4340 [5]

of events in the state machine. Further to this, pseudocode descriptions can be more expressive and explicit than some textual descriptions, since they provide an outline for implementations and therefore inherently require more detail. This increased explicitness, however, comes with the cost of reduced human readability.

As we have argued in this section, the differences across these methods are mainly syntactical and relating to levels of detail provided, but the state machines being described are conceptually identical. Because of this, we hypothesise that it is possible to develop a generalisable method for defining state machines for network protocols which is sufficiently expressive and flexible to encapsulate the different methods observed into one singular method. This common description method is beneficial since it allows for predictability in how state machines are defined in RFCs, in turn allowing for automated parsing of these.

Type of state machines for network protocols

A finite state machine is defined as a triple $M = (Q, \Sigma, F)$, where Q is a finite set of states, Σ is a finite set of inputs to the state machine, and F is a function $F : QX\Sigma \rightarrow Q$, representing transitions between states triggered by inputs [3].

A Mealy machine is a type of finite state machine which accounts for outputs as well as inputs. Therefore, a Mealy machine can be defined as $\hat{M} = (Q, \Sigma, \Theta, F, G)$, where Q, Σ and F represent states, inputs and transitions as previously defined, Θ represents the set of outputs, the output alphabet, and G is the output function $G : QX\Sigma \rightarrow \Theta$. This means that the outputs of the state machine are defined both by the current state and the input to the state machine [3].

A deterministic state machine is a state machine for which, given the same starting state and series of inputs, the same sequence of state transitions will occur.

Given these definitions, we argue that network protocols should be represented as deterministic Mealy machines. Firstly, in terms of determinism, it is important that network pro-

tocols are predictable. Whilst in networking certain events can be unpredictable, such as the loss of a packet or connection, the handling of the occurrence of these events should be predictable and repeatable to ensure the safety and robustness of protocols. This predictability allows for testing of the functionality of protocols, since outputs and transitions should be expected.

As an example to illustrate these concepts, we will show how the POP3 state machine can be represented as a deterministic Mealy machine. The state machine for POP3 can be defined as $\hat{M} = (Q, \Sigma, \Theta, F, G)$, where:

Q, the finite set of states, is the set:

{ DISCONNECTED, AUTHORIZATION, TRANSACTION, UPDATE, USER-ISSUED, USER-ACCEPTED, AWAIT-PASS-RESPONSE }

Σ, the finite set of inputs, which in the case of network protocols are known as events, is the set:

{ DELE [msg], LIST [msg], NOOP, PASS [pass], QUIT, RSET, RETR [msg], STAT, USER [name], GREETING, OK, ERR }

F denotes the function which, given the cartesian product of possible states and inputs, outputs another set of states, representing the state the machine transitions to given a starting state and an input. For simplicity, the following is a subset of the output of the function, and therefore is not the complete cartesian product of the set of states ad inputs listed previously:

$QX\Sigma =$

{(USER-ACCEPTED, PASS), (AUTHORIZATION, QUIT), (TRANSACTION, QUIT), (TRANSACTION, DELE), ... , (USER-ISSUED, OK)}

$Q =$

{AWAIT-PASS-RESPONSE, DISCONNECTED, UPDATE, TRANSACTION, ... , USER-ACCEPTED}

Where the order of these two sets represents the mapping of transitions. For example, if the state machine receives the input PASS whilst in the USER-ACCEPTED state, it will transition to the AWAIT-PASS-RESPONSE state.

Θ, the set of outputs, in the case of POP3 is the set of possible responses from the server upon the occurrence of an input:

{ POSITIVE-RESPONSE, NEGATIVE-RESPONSE, NUM-MESSAGES, SIZE-OF-MAILDROP, SCAN-LISTING, SEND-MESSAGE, DELETE-MESSAGE }

3. INCONSISTENCIES AND BARRIERS TO AUTOMATION

As has been discussed in Section 1, since most RFCs are written in natural language there is an issue with ambiguity and inconsistency in these documents. Specifically, in terms of state machines, there is no widely agreed-upon method for defining them in the standards community. As was shown in Section 2, there are several different methods used across RFCs to define state machines. Even in methods that are similar, subtle differences can be observed. While two RFCs may use the same method to describe their state machines, their formats often differ slightly. For example, in the case of ASCII diagrams, some protocols use boxes and capitals to denote states, while others omit boxes and simply contain text. These slight syntactical differences can be observed in Figure 4, with differences in how states, user calls and receiving and sending messages are represented.

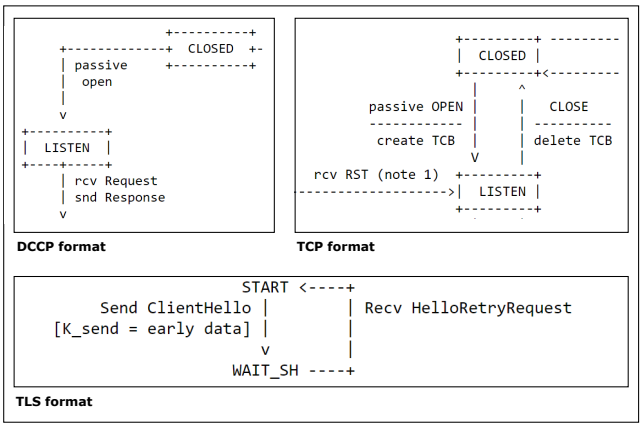
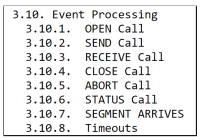


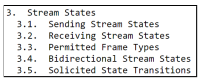
Figure 4: Comparison of different ASCII diagram formats across RFCs [5] [1] [14]

These slight differences are also present in RFCs that describe state machines textually. There are syntactical differences, such as the specific language used to denote a transition. For a state machine entering a "closed" state, for example, different representations of this transition can be "*the state machine enters the closed state*", "*after entering the closed state*" or "*the state changes to closed*", among others. Whilst the syntax varies slightly, conceptually these sentences are the identical. As well as these syntactical differences, there are also structural differences present in textual descriptions. A comparison of the sections for textual state machine descriptions of TCP, QUIC and DCCP is shown in Figure 5.

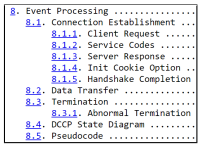
TCP provides a list of states and events that take place in the state machine, and in its event processing section lists



(a) TCP state machine section [1]



(b) QUIC state machine section [4]



(c) DCCP state machine section [5]

Figure 5: State machine definition sections across different RFCs

each event, followed by the states and a description of how this event affects each state. QUIC describes two state machines for the sending and receiving of streams, and is not as organised as TCP. Where TCP is structured and systematic, the QUIC RFC [4] narrates the functionality of the state machine in unstructured text. DCCP also contains a textual description of transitions and handling of events in different states which is not as systematically structured as TCP's event processing section, but also presents a structured pseudocode description of the state machine.

Whilst we have identified several different methodologies for defining state machines across RFCs, we argue that these methodologies conceptually define the same object. This can advantageous since it allows the incorporation of automated parsing of state machines, which in turn can provide benefits such as the generation of partial implementations and automated testing to prevent errors in the standards writing and implementation process. To achieve this, as illustrated in Figure 6, we propose the definition of a structured syntax for the definition of state machines in RFCs. However, a known issue in the incorporation of formal methods into the standards development process is that there are several barriers to their adoption. Other formal languages have been defined for different aspects of RFCs in order to avoid the issues caused by natural language, and these have not seen wide adoption by the community [18]. These barriers to adoption can be mitigated by designing formal methods which are sufficiently expressive whilst maintaining sufficient flexibility to be easily incorporated into authors' workflows [7]. To address this, in addition to the syntax proposed, we suggest the design of a standardised intermediate representation of state machines for network protocols. This allows for flexibility in the syntax used to describe state machines and the methods used for parsing to avoid overly constraining authors, whilst maintaining formality in the definition of these.

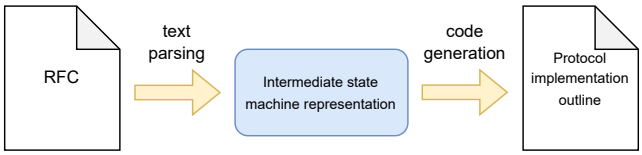


Figure 6: Process of compiling partial code implementations from RFCs

4. INTERMEDIATE REPRESENTATION OF STATE MACHINES

We have discussed previously the barriers to adoption that have been experienced in the standards writing community relating to the adoption of formal methods. To minimise adoption barriers, any formal methods introduced should be flexible and familiar, and this can be achieved by designing these formal methods to be as similar to those currently used as possible. While in this work we propose the development of a grammar that can be used to define state machines in text, we also emphasise the importance of a conceptual intermediate representation of state machines. This enables the use and development of other syntaxes which can be based on and parsed into this intermediate representation, removing the requirement to learn and use any specific syntax in order to benefit from the advantages of automated parsing. This makes the intermediate representation more accessible and flexible, addressing in this way the issues to adoption discussed. As well as this, a standardised intermediate representation allows for the output of a canonical representation of state machines. This means that as well as providing an abstract representation, the use of this abstract representation allows for the output of a specific format for network protocol state machines, which is machine readable and a standardised version of the state machine for a specific protocol. This allows for a point of comparison of state machines since they share the same format.

Existing representations of state machines

There is a variety of methods currently used for the representation of state machines. One method used to represent state machines is the use of formal description languages, such as Lucy [12] [10]. However, since Lucy can be used for the description of different types of state machines, it has a high level of abstraction. The language allows for the use of many features, such as symbols, guards, different types of states, actions and actors. While this is useful for a description that encapsulates different types of state machines, we argue that this level of abstraction is not necessary for state machines in the specific context of the description of network protocols. Other similar languages are Robot [11] and Frame [17], which have the same issue of complexity unnecessary for the context of this project.

Another method used for the description of state machines is via UML diagrams. Whilst UML diagrams are relatively simple to develop and read due to providing a visual overview of protocols, this comes with certain drawbacks. Firstly, diagrammatic descriptions of state machines are limited in the level of complexity they can express. As discussed in Section 2, several RFCs use ASCII diagrams for the representation of state machines, but these are often accompanied by some form of textual functionality description, stating that the diagrams are summaries and should not be interpreted as a full implementation. Secondly, UML diagrams can be used for a broad range of diagrams, and are not restricted to the description of state machines. This provides excessive flexibility for authors, which complicates parsing due to the allowance for slight representation differences. Further to this, the documentation for UML [2] is lengthy, and therefore complicates the learning process for authors. Whilst it is not necessarily required for an author to read the documentation to be able to develop a UML diagram, when arguing for formal methods it should be encouraged that these diagrams are represented in the same style, which is described through documentation. This learning curve once again raises concerns with the adoption of this methodology. Finally, the diagrammatic nature of

UML complicates machine readability of these representations, which is one of the goals of defining a standardised intermediate representation.

Because of these issues with other representations, we propose a representation that is sufficiently domain-specific to minimise the introduction of theoretical concepts. By only introducing concepts that are strictly necessary for the specific context of networking protocols, we constrain the description space enough to complicate the introduction of errors in how state machines are defined. As well as this, the representation must be sufficiently flexible to be easily incorporated into the current workflow of authors.

Design

State machines have a wide range of uses, and are not only used to describe network protocols. They can be used to describe a wide variety of systems, ranging from turnstiles and household appliances [6] to network protocols such as TCP. As we have briefly discussed in this section, limiting the context of state machine descriptions to that of network protocols can simplify the writing process for authors. To achieve this, we have designed a structured representation of state machines which limits its components to those in the specific context of network protocols. As we have discussed, RFCs use a range of methods to represent state machines, but their differences are mainly syntactical. We identify the following common conceptual components. A state machine to represent network protocols should be formed of three main elements: states, events and transitions.

- **States** represent the current status of the protocol, such as whether a connection is open, listening or closed.
- **Events** are the input to a state machine, representing an occurrence in the network protocol, such as the arrival of a packet or a timer going off.
- **Transitions** represent the movements between states caused by events. For example, when a server receives a specific packet which indicates it should close the connection, it may move from a state that indicates the connection is open to one indicating the connection is closed. Not all events cause a state transition, but transitions require the occurrence of an event.

These core elements of state machines can be observed in the ASCII diagram shown in Figure 7:

Using this representation, a state machine is formed of a set of states, a set of events and a set of transitions. States are represented by their name as is described in text. Events are also represented by their names, and they are categorised into types, which we identify to be API calls, incoming messages, outgoing messages and timers. Transitions are represented by references to relevant states and events. They contain a reference to the state the machine moves from, the state the machine moves to, and the event that causes this transition.

These elements are structured together to describe the state machine as follows. A state machine can be described as the set of all its states, which each are mapped to a set of transitions for which they are the state from.

In practice, this intermediate representation is developed using an object oriented approach, with classes representing

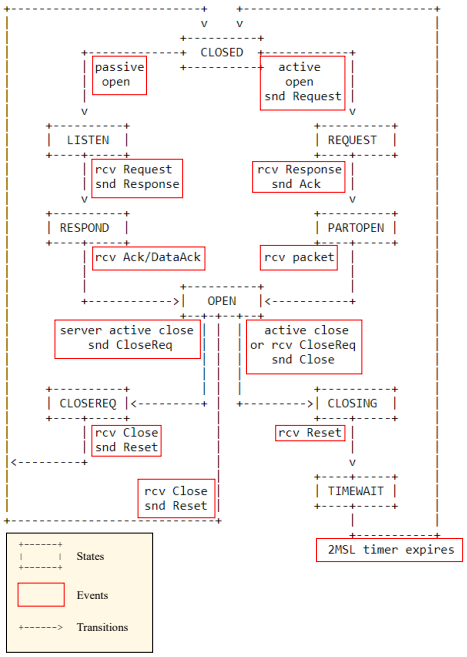


Figure 7: Annotated DCCP state machine diagram, adapted from [5]

the state machine, states, events and transitions. The state machine class contains a dictionary of state objects mapped to their relevant transitions, and these transitions contain references to the objects of the state from, state to, and event that causes the transition.

This representation facilitates a canonical output, in this case a JSON file where a state machine is formed of a dictionary of state names mapped to their relevant transitions, a dictionary of events mapped to their relevant types and a list of transitions, as shown in Listing 1.

Listing 1: JSON output

```
"state_machine" : {
    "states" : {"state_name" : [("
        state_name", "state_to", "event"])
        },
    "events" : {"event_name" : "event_type
        "},
    "transitions" : [("state_from", "
        state_to", "event")]
    }
```

5. SYNTAX FOR PARSING RFCS

Whilst we have discussed the importance of the design of an intermediate representation of state machines for network protocols, it is necessary to develop a methodology to parse the text found in RFCs in order to obtain this intermediate representation. This can be achieved using different methods, which is the benefit of having a standard intermediate representation, as it allows flexibility in terms of parsing. We have chosen to develop a structured textual syntax for this, identifying common patterns found in the current textual definitions of state machines across RFCs. We base our syntax on these definitions in order to maintain familiarity

to ease adoption.

TCP based structure

The structure used in the event processing section of the TCP RFC [1] is particularly systematic, which simplifies the design of a grammar to parse it. However, it maintains sufficient expressiveness to remain easily readable by humans. This syntax is explicit, as it lists and categorises all possible events and states that occur throughout the state machine. This can be useful for checking that all states defined are described in some way, and that no new unaccounted for states are defined throughout the event processing section. As well as this, the approach taken to describe event processing is systematic and predictable, following a pattern of listing events, each followed by all the possible states and these followed by a description of the processing that should take place. This processing describes the functionality of the protocol, but also defines the transitions of the state machine. This structure is illustrated in Figure 8.

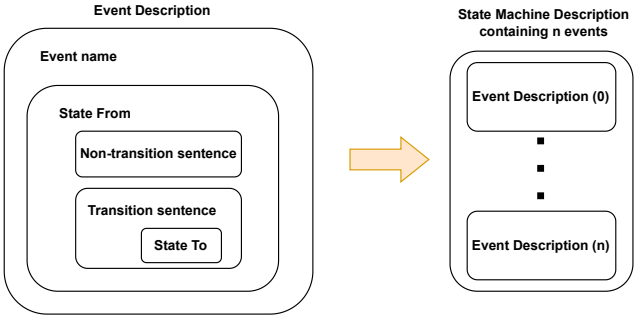


Figure 8: Structure of textual state machine description

The definition of transition and non-transition sentences allows for the identification of transitions that occur, whilst allowing for explanatory text to be included in the definition. This provides the flexibility required to allow for a syntax to be usable and readable. As long as events, the states from and the states to are clearly listed in our syntax, explanatory text can be present to describe the functionality of the protocol which may not be strictly necessary for the extraction of a state machine but is helpful for readers who are implementing the protocol. Furthermore, this structure follows the logical flow of the functionality of state machines. Events cause transitions, which is the order in which this syntax describes the state machine. This allows for the flow of the syntax to remain simple and readable, whilst simultaneously facilitating the generation of the intermediate representation as the text is parsed.

EBNF based grammar

To enable the parsing of the syntax previously discussed, we have developed an EBNF based grammar using Lark [16], a Python library which allows for text parsing. When parsing text, a tree is generated, which has as nodes each of the rules defined in the grammar. This allows for the traversal of the tree in order to retrieve the relevant elements from the text which are necessary to generate the intermediate representation.

The grammar defined expects a textual input of a state machine definition. This definition is formed of a listing of states, a listing of events and an event description. The two listings are structured as comma-separated lists, with state and event names and types written following a specific syntax defined, which can be observed in Figure 9.

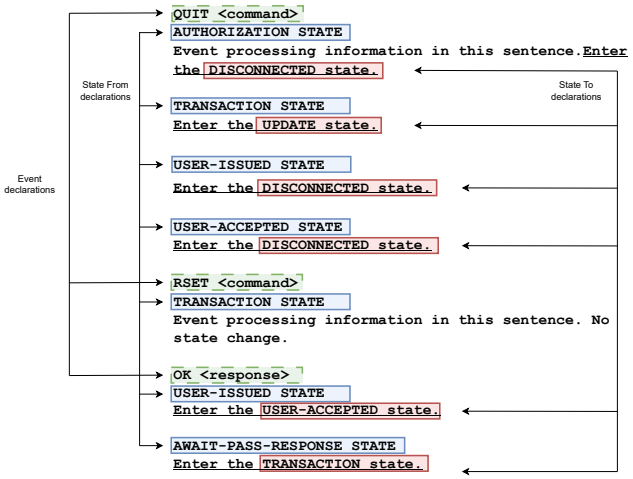


Figure 9: Part of POP3 state machine described in this syntax

The events description is the section of the state machine where transitions are defined. This is formed of smaller elements, as shown previously in Figure 8. A specific syntax is declared for the representation of event names, states from and transition sentences. As the text is parsed, these are represented into a tree format. This allows for a traversal of a the tree which follows the logical structure in which state machines are defined. An event description contains as its children subtrees of event names, which themselves contain subtrees representing state descriptions, themselves containing transition and non-transition sentences. This traversal allows for the generation of the intermediate representation.

6. CODE GENERATION

One of the benefits of automatically parsing state machines from RFCs is that the information retrieved can be used to generate partial implementations. These partial implementations can provide a structured guide for the development of full protocols. Given the development of a canonical representation of network protocol state machines, these can be checked for correctness using model checkers. This provides a level of confidence that the code generated from these models can be checked for logic errors, and therefore has the potential to provide guarantees around the introduction of bugs arising from these.

In this project, we generate partial implementations in Python, but given the nature of the intermediate representation these implementations could also be generated in other languages. The structure of the code generated is based on the intermediate representation discussed previously. A class is generated for each state, and within this class the events that cause transitions are represented as methods which return the state the machine transitions to given this event. This structure can be observed in Figure 10.

As well as reducing logic errors in implementations, the

```
class User_issued:
    def receivedQUIT_call():
        return Disconnected()


    def receivedOK_response():
        return User_accepted()


    def receivedERR_response():
        return Authorization()
```

Figure 10: Representation of a state in generated code for POP3

generation of code speeds up the development process for protocols, since the main logic of the implementation work is performed automatically, and development can focus on the addition of event handling code.

7. EVALUATION

In this section we will discuss how well the intermediate representation, syntax and code generation achieved in this project can represent state machines present in different network protocols, such as TCP, TLS 1.3, and BGP.

Intermediate Representation

The intermediate representation designed in this project represents state machines as a set of states and transitions, the latter formed of groupings of events and states. This is simple, but sufficiently general to represent most state machines found in RFCs.

When using the syntax and intermediate representation defined in this work to represent state machines in existing RFCs, the following observations are made:

Firstly, the argument we make for singular events occurring in one state causing one transition is not the case in the description of certain protocols. As an example, we can analyse at how TCP [1] describes the handling of a segment arrival. In its syntax, "SEGMENT ARRIVES" is defined as a singular event. Following our pattern for describing state machines, this event should be sufficiently informative to trigger at most one transition for each state. However, the handling of this event is in reality more complex, and depending on different factors of the TCP session different transitions are possible. In the handling description of this event when the session is in the "SYN-SENT" state, there are several potential transitions described depending on the settings of different bits present in the segment. One potential transition described is that if the RST bit is set and the ACK is acceptable, the state machine moves to the "CLOSED" state. Another possible transition occurs during the checking of the SYN bit, which only occurs if the ACK is acceptable or there is no ACK and the segment does not contain a RST. Further checks are performed, with the state machine potentially moving to the "ESTABLISHED" state depending on their results. Given other conditions dependent on similar checks to those mentioned here, the state machine may also move to the "SYN-RECEIVED" state. This description violates the principle of the determinism of network protocol state machines, since given the same event and transition the machine will not always perform the same transition, as it depends on factors relating to the segment being received.

From this we can conclude that events require a more rigorous description to account for these factors that affect transitions. This may involve splitting this event into smaller, more explicit events that account for other factors which affect state transition. For example, the event "SEGMENT ARRIVES" should be expanded to describe "SEGMENT ARRIVES + RST BIT SET" as well as the other factors that differentiate which state the machine transitions to discussed previously.

Secondly, some transitions are dependent on a certain number of events taking place, and the transition is not valid until all these events have occurred in order. An example of this is TLS 1.3 [14], as shown in Figure 11

```
|               WAIT_CV
|                  | Recv CertificateVerify
+> WAIT_FINISHED <+
       | Recv Finished
       | [Send EndOfEarlyData]
       | K_send = handshake
       | [Send Certificate [+ CertificateVerify]]
       | Send Finished
       | K_send = K_recv = application
       v
   CONNECTED
```

Figure 11: Part of TLS 1.3 client state machine, highlighting multiple events leading to one transition [14]

This is an aspect that the current intermediate representation does not handle. To deal with this, intermediate states could be represented, splitting the transition currently defined as a singleton into smaller states that reflect the occurrence of each event required to arrive at the end state.

Thirdly, most protocols contain optional features. POP3 allows for several authentication methods to be used, TLS 1.3 has optional modes such as 0-RTT/No 0-RTT or the use of PSK/Certificates, and BGP [13] provides several optional events allowing for different functionalities. With the current design of the intermediate representation, optional events are not handled. However, this becomes a complex issue, since optional events are not necessarily optional in the sense that they are not required for the basic functionality of the protocol. POP3 defines different methods for authentication, all described as optional. However, this does not imply that the authentication process itself is optional, simply that the specific method used can vary between implementations. This highlights that a more explicit definition of optional events and their requirements is necessary.

Finally, our intermediate representation categorises events into the types: API calls, incoming messages, outgoing messages and timers. This is generally descriptive enough for transport protocols, but for example, BGP [13] presents its own categorisation of events: Administrative Events, Timer Events, TCP Connection-Based Events and BGP Message-Based Events. These categorisations can be adapted to our

type descriptions, with administrative events belonging to a similar categorisation as API calls, and TCP Connection-Based as well as BGP Message-Based events describing incoming and outgoing messages. However, given that an explicit difference is made between different types of messaging events is made, it may be beneficial to provide more flexibility to authors and allow for an optional configuration in the intermediate representation that allows for custom event types being defined.

Syntax

Since the intermediate representation we have defined for the encoding of state machines is built of simple elements, this allows for the syntax to be simple and easy to use. Keywords are used for the definition of states, event titles and the representation of transition sentences. This allows for definitions of state machines with little addition of formality to the methods already used, requiring only slight changes to the format of state and event declarations as well as transition sentences. Furthermore, the allowance of non-transition sentences to be present in the event handling description maintains the human readability of these definitions, and this text can aid in the understanding of the functionality of the state machine. This achieves the required balance between machine and human readability.

Currently the syntax used supports one format for transition sentences, of the form "*enter the STATENAME state*". Other RFCs contain slight variations of this, such as "*the state changes to the STATENAME state*" or "*after entering the STATENAME state*". Because of this, a translation to our format is required, which is a barrier to adoption. However, the addition of these options to the grammar is straightforward, as the grammar supports the addition of further types of transition sentence types. Therefore, although currently translation is required, small additions can be made to the grammar to reduce the effort of these translations.

Finally, as has been discussed in Section 5, the structure of the syntax defined is based on TCP's event processing section, defining state transitions relative to events and how these affect the states of the machine. Other RFCs do not follow this structure, but do have similar structures of their own. For example BGP and POP3 follow a parallel structure to TCP, describing their state machines relative to states rather than events. The difference in this structure is that states are listed instead of events, and following the state listing each event that affects that state is listed and described, declaring the transitions caused. Logically, both of these structures are similar and translation between the two is not complicated. However, given the similarity of this structure style to that of TCP, it would be beneficial for the usability of the syntax to add support for this structure style.

Code generation

The development of a syntax and an intermediate representation achieved in this project allow for parsing standards documents and the generation of partial code implementations. These implementations can aid developers in the implementation of protocols, providing structure based on their state machines.

The code generated in this work is a general structure of the states present in the protocol and the events that trigger transitions between these states. Event handling is largely left to the developer, with the code generating providing an outline of where in the state machine events take place and the state the machine moves to. This allows for flexibility in the definition specific details which can vary across different implementations, whilst still ensuring that the logical structure of the protocols is identical. Due to the limitations of the intermediate representation mentioned previously, the complexity of the code generated is also limited. Additions to the syntax and intermediate representation could allow for more detailed code which handles state variables, conditionals and outputs more robustly. However, the trade-off of this increased complexity in the syntax is that it requires increased formality, which can reduce its usability.

8. RELATED WORK

In this section we discuss some of the work relating to the issue of the use of natural language for the definition of network protocols. While different methods are used within the field to approach this issue, there is a general agreement that reducing ambiguities and inconsistencies in standards documents is becoming increasingly necessary. Zave et al. [20] discuss in detail the inadequacy of informal methods to describe protocols, and suggest transitioning towards more formal descriptions. These, however, are not required to be fully formal language descriptions, and emphasis is drawn to the utility of natural language based on formal models that can be understood by those unfamiliar with formal language.

McQuistin et al. [8] propose a standardised description language to enable automatic code generation for parsing network packets. They discuss the tradeoff between the technical and social aspects of adopting a standardised language, as it must be strict enough that documents can be parsed, but simultaneously not so strict that they have a formality unusable by those who are not experts in formal methods . They present the Augmented Packet Header Diagrams protocol description language, designed to represent packet header diagrams. This syntax is machine-readable, but also similar in format to the diagrams currently used in RFCs. However, a problem with this approach is that it only parses and presents a structure for packet header diagrams. Therefore, it does not have support for state machines and consequently has some functionality limitations, such as checking the order in which certain packets arrive.

Another approach taken towards automated code generation is that of Yen et al. [19]. They develop SAGE, a natural language processor that parses standards documents. The aim of this research is to understand the semantics of a specification and from these semantics automatically generate code. This research addresses the issue of ambiguity in the use of natural language for protocol specifications. In the process of parsing specifications, SAGE highlights any ambiguities found and allows the user to disambiguate these. Once no ambiguities remain, it is capable of generating code. However, SAGE is limited in what it can parse. Currently, it only fully supports parsing the syntax of header diagrams and listings, but not yet that of state machine diagrams.

Although Yen et al. do not yet provide functionality for parsing state machine diagrams, Pacheco et al. [9] propose a finite state machine (FSM) extraction method, using a data-driven approach. By having access to state machines that describe network protocols, certain techniques can be used

to check that they function correctly and safely. Because of this, Pacheco et al. propose this automated extraction from RFCs. They show that their approach generalises to several protocols, generating FSMs for BGPv4, DCCP, LTP, PPTP, SCTP and TCP. However, it is noted that it is currently not possible achieve a full translation between canonical FSMs and those that are generated by their approach due to ambiguity issues. Therefore, the state machines generated will not always reflect the correct behaviour of a protocol.

9. CONCLUSIONS

The use of formal methods and natural language for the definition of network protocols each have advantages and disadvantages. Mainly, there is a trade-off in the placement of workload. Whilst natural language is simplifies the authorship process, a larger workload is present when dealing with the errors and ambiguities inherent to this method. Formal methods on the other hand allow for automation and testing after the definition of standards, but can often have a steeper learning curve. In this project, we motivate the incorporation of usable formal methods into the standards writing process, aiming to strike a balance between formality and familiarity.

We have shown that there are observable patterns in state machines present in RFCs, and that these patterns, whilst syntactically different, represent the same types of state machines. We develop a common method for describing network protocol state machines, which maintains similarity to the process currently used whilst allowing for automated parsing, in turn allowing for the generation of partial implementations.

Whilst there are limitations to the complexity this description of state machines can represent, we have shown that with further expansion of the syntax and more clear definitions of the components of state machines in RFCs, it is possible to use a common structure for their representation.

Acknowledgments. I would like to thank my supervisors, Colin Perkins and Stephen McQuistin, for the feedback and support they have provided me with throughout this project.

10. REFERENCES

[1] W. Eddy. Transmission Control Protocol (TCP). RFC 9293, Aug. 2022.

[2] O. M. Group. Unified modeling language, December 2017. Accessed on March 9, 2023.

[3] M. Holcombe. *Machines and semigroups*, page 25–75. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1982.

[4] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[5] Kohler, E., Handley, M., and S. Floyd. "Datagram Congestion Control Protocol (DCCP)". RFC 4340, March 2006.

[6] T. Koshy. Formal Languages and Finite-State Machines. In *Discrete Mathematics with Applications*, pages 733–802. Elsevier, 2004.

[7] S. McQuistin, V. Band, D. Jacob, and C. Perkins. Parsing Protocol Standards to Parse Standard Protocols. In *Proceedings of the Applied Networking Research Workshop on ZZZ*, pages 25–31, Virtual Event Spain, July 2020. ACM.

[8] S. McQuistin, V. Band, D. Jacob, and C. Perkins. Investigating Automatic Code Generation for Network Packet Parsing. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–9, Espoo and Helsinki, Finland, June 2021. IEEE.

[9] M. L. Pacheco, M. von Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru. Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents, Feb. 2022. arXiv:2202.09470 [cs].

[10] M. Phillips. Lucy docs. Accessed on March 9, 2023.

[11] M. Phillips. Robot library documentation. Accessed on March 9, 2023.

[12] M. Phillips. Announcing lucy, April 24, 2021. Accessed on March 9, 2023.

[13] Y. Rekhter, S. Hares, and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Jan. 2006.

[14] E. Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.3". RFC 8446, August 2018.

[15] D. M. T. Rose and J. G. Myers. Post Office Protocol - Version 3. RFC 1939, May 1996.

[16] E. Shinan. Lark parser documentation, October, 2021. Accessed on April 19, 2023.

[17] M. Truluck. Frame a software architecture language. Accessed on March 9, 2023.

[18] J. Yen, R. Govindan, and B. Raghavan. Tools for disambiguating RFCs. In *Proceedings of the Applied Networking Research Workshop*, pages 85–91, Virtual Event USA, July 2021. ACM.

[19] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 272–286, Virtual Event USA, Aug. 2021. ACM.

[20] P. Zave, T. Laboratories, and F. Park. Experiences with Protocol Description. page 6.