Modeling and Testing Implementations of Protocols with Complex Messages

Tom Tervoort and I.S.W.B. Prasetya

Dept. of Inf. and Comp. Sciences, Utrecht University, the Netherlands
s.w.b.prasetya@uu.nl

Abstract. This paper presents a new language called APSL for formally describing protocols to facilitate automated testing. Many real world communication protocols exchange messages whose structures are not trivial, e.g. they may consist of multiple and nested fields, some could be optional, and some may have values that depend on other fields. To properly test implementations of such a protocol, it is not sufficient to only explore different orders of sending and receiving messages. We also need to investigate if the implementation indeed produces correctly formatted messages, and if it responds correctly when it receives different variations of every message type. APSL's main contribution is its sublanguage that is expressive enough to describe complex message formats, both text-based and binary. As an example, this paper also presents a case study where APSL is used to model and test a subset of Courier IMAP email server.

1 Introduction

Modern communication protocols are often quite complex. Implementing one is always tricky and error prone, and therefore an implementation should be thoroughly tested before it is used. The complication lies not only in the interactions between the communicating parties, but also in the format of the messages. A message can be a quite complex record structure with multiple fields, some could be optional, and some could have delicate dependencies, which in turn are prone to errors. Testing a protocol implementation can be made much more systematic if people first construct a formal model of the protocol. In practice, people often do not do this. They derive test cases directly from the natural language document that describes the protocol. Although there are standards for such documents, e.g. RFC [30], and guides on how to write a good description [32], a natural language description can still be deceiving and ambiguous.

Attempts to provide formal languages to model protocols have been mostly focused on describing the interaction parts, e.g. the languages SDL [5], Estelle [7], and even UML [23] (in particular the MSC part of UML). A protocol is typically described as a system of participating actors, each is described by some form of labelled transition systems. However, these languages ignore the

complexity of message formats, which make them not really usable for describing many real world protocols with complex messages. Model-based automated testing tools for protocols, e.g. TorX [38,17], TGV [15], and Phact [20], follow the same trend. These tools can generate test cases from a formal model of a protocol, but they too focus on testing the interaction aspect of the protocol. So, conformance validated by these tools does not imply that we have covered all possible message structures. For example, we may still be left uncertain whether an implementation under test would respond correctly if it gets a message with a certain optional field voided.

To address this gap, this paper contributes a language called APSL (A Protocol Specification Language) to formally describe how the messages in a protocol are formatted and how parties in the protocol interact by exchanging these messages. APSL tries to be a light weight language while still be expressive enough to describe real world text-based as well as binary protocols. A text-based protocol is a protocol whose messages are human readable strings, whereas the messages of a binary protocol consist of low level bitstrings. FTP, SMTP, and IMAP are examples text-based protocols. WebSocket is an example of a binary protocol.

From an APSL description of a protocol, an implementation of different parties in a protocol can be automatically tested. Under the hood, APSL provides two basic testing functionalities: (1) its generator can randomly generate messages of a *correct format*, and (2) its parser will check if the messages sent by an Implementation Under Test (IUT) have a correct format. Just having these functionalities saves testers much effort —else, they would have to write a custom message generator and parser for each protocol they test, or to find a third party dedicated generator and parser for the protocol. APSL's test engine is built on top those functionalities. When invoked invoked, it will automatically traverse the IUT's interaction model and in doing so test the IUT. To provide flexibility, the engine can be parameterized by a traversal strategy to control how the IUT is to be tested.

This paper is structured as follows. Section 2 introduces APSL. Due to limited space, we will only present a subset of APSL. A description of its full syntax can be obtained from [34]. Then, Section 3 explains how APSL test engine works. We have studied several examples to investigate the feasibility of using APSL for testing real world protocols. Section 4 will highlight one of them: we show that we can use APSL to describe a subset of the IMAP protocol and test a server-side implementation of it. Section 5 discusses related work. Section 6 concludes and discusses some future work.

2 Describing Protocols in APSL

To describe a protocol in APSL we define a *message module* and an *interaction module*. The first describes different types of messages that the protocol use, and how they are formatted. The second abstractly describes which *actors* (parties) take part in the protocol and how they interact by exchanging messages.
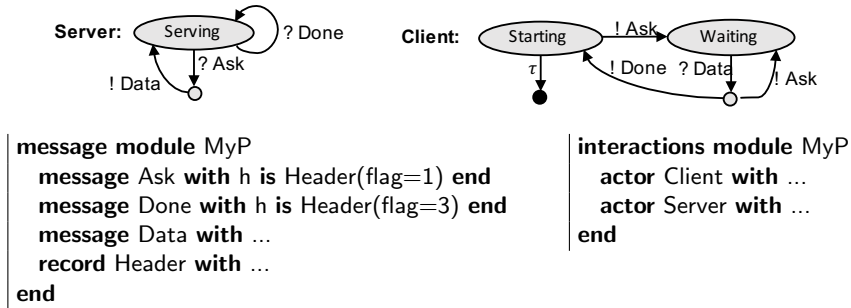
Fig. 1. MyP protocol, with two actors, a server and a client, and its top level APSL description.

As a running example, consider a protocol called MyP shown in Figure 1, with a server and a client as actors, interacting as visually shown by the corresponding Labelled Transition Systems (LTS's). A transition in the LTS of an actor represents either an action by the actor to send a message (denoted by '!'), or the receipt of a message (denoted by '?'), or a non-observable internal action by the actor (denoted by τ). Each send and receive transition is labelled by the type of the message that is to be sent or received. There are three types of messages in the protocol. 'Data' messages are sent by the server to the client; they carry payload. The client sends an 'Ask' message to the server to ask for an instance of Data, and a 'Done' message if it decides that it has enough.

Figure 1 also shows the top level APSL description of MyP. The message module declares the above mentioned three types of messages, whereas the interaction module declares the two actors (server and client). Messages are basically records. A *record* is composed from *fields*, which comprise the core building blocks of an APSL specification. In the example, the message types Ask and Done are both defined to have a single field named h of a type called Header.

2.1 Specifying complex messages

Each field in a record has a name and a type, e.g. Integer or Text. In typical specification languages, e.g. OCL [8] or Z [33], we do not have to care about how types are implemented. In protocol engineering we need to. Ultimately, messages are exchanged in bits. A protocol may insist on a specific way in which values are encoded in bits. E.g. it may require a certain integer field to be represented in a 4-bits big-endian, while another integer field should be 32-bits, and so on. So in addition to specifying the type of a field, in APSL we also need to specify a so-called *codec* to describe how instances of the type should be formatted/represented in bitstrings. The general syntax of a field declaration is:

$$fieldname \textbf{ is } type \textbf{ as } codec$$

APSL comes with a range of common codecs. For example a codec called BoolBits is used to translate boolean true and false to specific bit patterns. The codec

TerminatedText can be used to format a variable length text. It additionally appends a terminator to indicate where the text ends. Alternatively, if the length of the text is fixed, the codec FixedCountText can be used (when this codec is used, the receiver is thus assumed to know where in the received bitstring such a field ends). Almost all codecs have parameters to further specify its format. E.g. BoolBits requires us to specify the bit patterns to use to represent true and false, whereas all text codecs such as FixedCountText require the used character set to be specified.

As an example, below we define the record type Header, to be used in the Ask and Done messages of MyP. We define it to be a record with two fields:

```
record Header with
  flag is Integer as BigEndian(signed=false,length=2)
  reserved is Binary(value=b'000000')
end
```

The field flag is of type integer. The codec says that the integer will be represented by two bits in the unsigned big-endian format. The field reserved is of type Binary, which means it is simply a bitstring. For such, no codec is required.

Types can be primitive (such as Integer), records, or user defined types. APSL also supports variant records (union types) —see the documentation in [34]. APSL supports *dependent types*. These are types that are parameterized by value-level expressions used to specify a subset of such a type. For example, the type expression Binary(value=b′000000′) above specifies a subset consisting of a single value, namely the bitstring 000000 (so, this is the only allowed value of the field reserved). As another example, Integer(min=0, max=500) specifies a subset of integers, from 0 up to 500.

An important principle in protocol design is that the receiver of a message should be able to efficiently determine when a message ends, and similarly, when each field within a message ends. One way to achieve this is to end a field with a specific bit pattern (e.g. as in the TerminatedText codec). Another common convention used in various protocols is to have a field that specifies the length of the next field or fields. Dependent types are essential to capture such dependency. This is shown by the example below that defines a record type called DataItem:

```
record DataItem with
  n is Integer(min=0,max=500) as BigEndian(signed=false,length=32)
  data is Binary(length=8*n)
  padding is Binary(length=8*(4 − n%4), char8_pattern=/\0*\1/)
end
```

A DataItem basically carries some binary data. The length of this data is encoded in the field n, which should be an integer between 0 and 500. Its corresponding codec specifies how this n is encoded in bits. The field data contains the actual data, its type parameter specifies that its length should be exactly 8n bits. The field padding is more complicated. Sometimes, a protocol requires that a message, or a part of the message, to have a length which is a multiple of a certain number. Suppose we want the total length of a DataItem to be a multiple

```
interactions module MyP
actor Client with
  init state Starting where anytime do send Ask next Waiting or do quit end
  state Waiting where on Data do send Ask continue
                           or do send Done next Starting end
end
actor Server with
  init state Serving where on Ask do send Data continue
                      on Done do continue end
end
```

Fig. 2. The full interaction module of MyP.

of 32 bits. The field padding is used to pad it if that was not the case. The type
parameter $\mathsf{length} = 8 * (4 - \mathsf{n}\%4)$ specifies how long the padding should be. The
type Binary can also be parameterized by a *regular expression* to specify allowed
bitstrings. Above, the regular expression in $\mathsf{char8\_pattern} = /\backslash0*\backslash1/$ specifies that
a bitstring of zero or more 0's closed by a 1 should be used as the padding.

For MyP, we still have to define the message type Data. The definition below
shows an example of a more complicated message type involving optional fields.
Let's define the type Data whose instances are records that contain a header and
an optional footer. The latter is specified by the type Optional, which is another
example of a dependent type: the occurrence of the footer is made to depend
on the boolean value of another field, namely hasfoot (a footer exists of hasfoot
contains a value that represents true, else there is no footer).

```
message Data with
  h is Header(flag=0)
  payload is List(elem=DataItem,max_length=4)
          as CountPrefixList(count_codec=Word32Codec)
  hasfoot is Bool as BoolBits(falsehood_string=X'00',truth_string=X'ff')
  foot is Optional(is_empty=!hasfooter,subject=Text) as ... # some codec
end
```

Instances of Data also contain a field called payload, which is a list of up to four
instances of DataItem (defined above) as the messages' payload. APSL provides
several standard codecs for translating a list (such as the field payload above)
to bitstrings. The codec CountPrefixList used above will prepend the list with an
integer stating the length of the list and then sequentially write out the items of
the list to bitstring.

2.2 Specifying interactions

In APSL, the interaction between the actors of a protocol is described in an
interaction module, by modeling each actor as an LTS a la [37] —more precisely,
as an input-output LTS (IOLTS) [15], which is an LTS where we distinguish be-
tween send and receive actions. The IOLTS describes how the actor exchanges
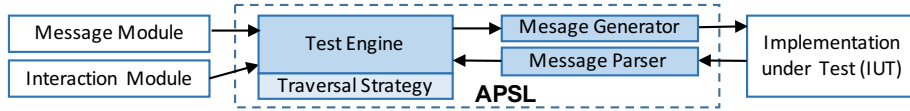
Fig. 3. The architecture of APSL automated testing.

messages with its environment. The latter represents, from the actor's perspective, all the other actors. The top level of the interaction module for the example protocol MyP was shown in Figure 1. Now, Figure 2 shows the full code of this interaction module; it reflects the graphical IOLTS in Figure 1. A send M expression in an actor's description represents an action by the actor to send a message of type M to the environment, whereas an on M expression represents the receipt of a message of type M from the environment. For example, the module in Figure 2 states that the Client has two named states, Starting and Waiting; the first is also the initial state. When in the state Starting, Client can at any time either quit, or send an Ask message (an APSL model can thus be non-deterministic). If it does the latter, it will also move to the state Waiting. If it receives a Data message while in the state Waiting, it will either send another Ask, or send a Done message, after which it moves back to the Starting state.

Let's call the APSL description of an actor an *interaction model* of the actor. Being a pure LTS, such a model only describes how the behavior of the actor depends on its LTS state and on the current message *type*. In particular, the model does not specify what the content of the message should be, nor how the behavior of the actor depends on this content. For example, when the Client of MyP receives a Data it can either respond with an Ask or a Done. The decision could depend on the content of Data. This is however abstracted away from the model in Figure 2. To some extent, such dependency is still expressible by an LTS, namely by defining different message types to represent instances of Data with different kinds of content. This works as long as the number of the latter is finite.

3 Testing

Figure 3 shows the architecture of APSL automated testing framework. Given a pair of message and interaction modules that describes a protocol, APSL test engine can automatically test implementations of the protocol's actors. The actors are tested one at a time. Consider an Implementation under Test (IUT) I_A that implements an actor A in the interaction module. When invoked, the test engine generates a test in the form of a traversal over the IOLTS induced by A. So essentially, APSL does model-based testing. The test engine takes the role of A's environment. A traversal starts at A's initial state, and step-wisely extends itself by following a transition to the next state. If the transition requires a message of some type M to be sent to the IUT, the test engine will invoke the message generator to produce a random but correctly formatted message of

```
 1: procedure test(A : Actor)                    ▷ the top level procedure to test an actor
 2:     traverse(A, τ*({ A's initial state }), [ ])              ▷ generate a traversal
 3: end procedure
 4:
 5: procedure traverse(A, S, trace)              ▷ S is the set of possible current states
 6:     if some end condition then throw Done
 7:     status ← parse(receive())
 8:     case status of
 9:         ValidFormat : msgType ← getMsgTypeOf(getMsg(status))
10:         InvalidFormat : throw invalidFormatError
11:         TimeOut :                              ▷ the IUT did not send anything
12:         do
13:             ▷ calculate which types of messages can currently be sent to the IUT:
14:             V ← { α | s∈S ∧ (∃t. s →?α t) }
15:             if V=∅ then traverse(A, S)
16:             ▷ invoke a 'strategy' to decide which message type to send:
17:             msgType ← strategy(S, V)
18:             ▷ generate an instance of the message type, and send it to IUT:
19:             send(generateMsgInstance(msgType))
20:         end
21:     end
22:     trace ← trace.addLast(msgType)                         ▷ extend the trace
23:     S ← { t | s∈S ∧ s →msgType t }             ▷ calculate the possible next states
24:     if S=∅ then throw invalidTraceError(trace)
25:     traverse(A, τ*(S), trace)                  ▷ recursively extend the traversal
26: end procedure
```

Fig. 4. *APSL's traversal algorithm. In the algorithm, $s \xrightarrow{a} t$ means that the actor A contains a transition from the state s to t, labelled with a. The expression $\tau^*(S)$ means the set of states that can be reached through zero or more τ-transitions from some state in S. This set always includes S itself.*

type M, and send it to the IUT. Conversely, if the transition is triggered by the receipt of a message from the IUT, the message parser will first check if the message has a valid format.

In model-based testing it may seem sufficient to just generate model-level test cases. However, note that to actually test an IUT we still need a message generator and a message parser to convert model-level test cases to concrete interactions with the IUT. These components are tedious and error prone to write. Fortunately, they are built in APSL, without which testers will have to write them themselves for each protocol they test.

Figure 4 shows APSL's traversal algorithm. The algorithm keeps track of the trace of the types of the messages exchanged with the IUT so far. It also keeps track of the actor's current state, so that it knows which transitions can be traversed to get to the next state. Since an APSL actor can be non-deterministic, the algorithm may not be able to uniquely determine IUT's current state. It therefore maintains a set S of of possible current states. If σ is a trace of message

types, let $s \overset{\sigma}{\Rightarrow} t$ means that there exists a path in the actor's LTS, to go from the state s to the state t, such that if we remove all the τ labels from the trace of labels induced by this path, the trace is equal to σ. Let s_0 be the actor's initial state, and suppose S is the set of current states according to the traversal algorithm, and *trace* is the traversal's trace so far. The algorithm maintains the following property of S:

$$S \;=\; \{\, t \mid s_0 \overset{trace}{\Longrightarrow} t \,\}$$

If no error is found, line 6 decides when the traversal is ended, e.g. it could impose a limit of N transitions. When in the current states there are multiple receive transitions possible (note that these are thus transitions where the test engine can send a message to the IUT), the algorithm invoke a 'strategy' to chose which one to follow (line 17). The default strategy simply randomly chooses a transition, thus inducing a random traversal over the LTS. APSL provides a hook that allows this strategy to be redefined.

During a traversal, the test engine checks for two correctness aspects: (1) the IUT should produce messages that conform with the format defined in the message module (line 10), and (2) the trace of the messages exchanged with the IUT should conform with A (line 24). The latter means, more precisely, that:

$$\{\, t \mid s_0 \overset{trace}{\Longrightarrow} t \,\} \;\neq\; \emptyset$$

The engine itself will not send a message that would violate the trace conformance, so if it is violated, the IUT is to blame.

At the end of a traversal, the test engine reports two kinds of coverage: the coverage over the transitions IUT's LTS model, and the coverage over the messages. For the latter, the engine checks if every field (of every message type and the underlying record types) and every value of every enumeration type has been tested. Recall that fields are declared in a message and record definitions. In addition, APSL also supports variant records and enumeration type (not shown in the paper). A field is considered as covered by a traversal if it ever occurs in some message during the traversal. For an optional field, its absence also counts a coverage goal.

4 Case Study

To investigate whether APSL is expressive enough to describe and provide at least basic automated testing of implementations of real world protocols we conducted two case studies: Courier email server [13] and Autobahn WebSocket server [1]. On a smaller scale, we have also used APSL to describe NTP time format [26], BSON data format [6], and DNS message format [36].

This section will present selected highlights of the email server case study. The full account of the case study, as well as that of the Autobahn WebSocket case study, can be found in [35]. Other mentioned examples can be found in APSL's home [34]. A Courier email server is used to store emails. Clients can

connect to it to access and manage the stored emails. The server implements the Internet Message Access Protocol (IMAP) protocol, more precisely version 4rev1 defined in RFC 3501 [11], that defines how an IMAP server should communicate with its clients. The following subset of the protocol is considered in the case study:

- The client can log-in with a username and password.
- In reply, the server greets the client. There are three different greetings, to indicate that the user is either required to log in, or has already been authenticated, or has no access.
- The client can examine, create, delete and rename mailboxes.
- The client can select a mailbox, after which emails within can be fetched, altered or copied. Entire emails can be fetched, or only their metadata, or only their size and flags (e.g. whether an email has been read). The client can close the mailbox, allowing the client to continue opening another.

In this case study we want to know whether APSL is able to describe the part of the IMAP's server side protocol that covers the above functionalities. Section 4.1 below shows that the answer is "yes". Then, we then want to know if APSL can actually be used to test the Courier server; this is discussed in Section 4.2.

4.1 Modeling IMAP

Client commands Within IMAP, messages sent from the client to the server are called *commands* and messages from the server are called *responses*. A command starts with a tag, an arbitrary string of alphanumeric characters chosen by the client, followed by the command name, and the command's arguments whose types depend on the command. These parts are separated by a single space, and the command always ends with a line terminator. Since the first two fields are the same for every command, we can describe them as follows in APSL:

```
record CommandStart(commandName) with
  tag is Tag as SpaceTerminated
  name is Identifier(value=commandName) as SpaceTerminated
end
```

APSL supports user defined types and codecs. Tag and Identifier above are user defined types, whereas SpaceTerminated is a user defined codec. Their definitions:

```
type Identifier is Text(charset='ascii', pattern=/[!−˜]+/,
                        exclude_pattern=/ |\r\n|\*/,max_count=20)
type Tag is Identifier(pattern=/[0−9a−zA−Z]+/)
codec SpaceTerminated is TerminatedText(encoding='ascii',terminator=' ')
```

An Identifier is thus a text of maximum 20 characters which should not contain any space, line break, nor the '*' character, whereas a Tag is a purely alphanumeric Identifier. The SpaceTerminated codec formats a given text by appending a single space to its end.

The are 11 commands that correspond to the previously listed IMAP functionalities. They all start with a CommandStart followed by one or more command specific arguments. For example, the structures of the command to delete a mailbox is shown below:

```
message DeleteCmd with
  _start is CommandStart(command='DELETE')
  mailbox is MailboxId as LineTerminated
end
```

The last field in a command must be terminated by a line break. This is specified by the LineTerminated codex, whose definition is analogous to SpaceTerminated.

The field mailbox in the DeleteCommand poses however a challenge for automated testing. It contains a value of the type MailboxId, which is text representing the name of an *existing* mailbox. APSL's message generator will have a very hard time generating such a name. Any generic generator will have the same problem. Fortunately, this can be mitigated in APSL over-specifying the type MailboxId, e.g. as follows: **type** MailboxId **is** Identifier(pattern=/INBOX|NOBOX/).

Server responses After receiving a command, the server answers with a tagged status response to indicate whether the commanded operation succeeds. This status response may however be preceded by a number of untagged responses providing some relevant information depending on the command. A *tagged status response* starts with a tag, followed by a status text indicating the result of the triggering command, and another text with additional information. There are three types of statuses to indicate: (1) the received command was successful, (2) the command failed, and (3) the command was incorrectly formatted. As before, fields should be separated by a single space; the last one should end with a line break. The formalization in APSL is shown below. We capture the three types of status responses with three different message types, each is just a wrapper over the underlying StatusResponse record type that contains the actual fields mentioned above.

```
message OkResp with resp is StatusResponse(response=ok) end
message NoResp with resp is StatusResponse(response=no) end
message BadResp with resp is StatusResponse(response=bad) end
record StatusResponse(response) with
  tag is Tag as SpaceTerminated
  _id is StatusResponseId(value=response) as SpaceTerminated
  text is Text(charset='ascii', pattern=/[ -~]*/, exclude_pattern=/\r\n/)
      as LineTerminated
end
enum StatusResponseId of Text with ok as 'OK' no as 'NO' bad as 'BAD' ... end
```

There are actually two additional statuses not shown above: PREAUTH (a server greeting indicating the client is already authenticated) and BYE (sent when logging out). These are used in certain untagged responses.

As said, the server may precede a tagged status response with a series of untagged responses. There are two types of untagged responses: those that also report some status, and those that do not. Untagged status response has the same structure as a tagged response, except that the tag is replaced by a '*'. It may report an intermediate status of an operation. In addition to OK or NO statuses, some may report PREAUTH or BYE statuses. Non-status untagged responses always start with a message number (since they always relate to a specific e-mail), followed by an identifier indicating their type. They are expressed by wrapping a message type around the underlying record type below:

```
record UntaggedResponse(kind) with
  _asterisk is Text(value='*') as SpaceTerminated
  msg_id is MessageId as TextInteger(text_codec=SpaceTerminated)
  response_type is Identifier(value=kind) as SpaceTerminated
  info is Text(charset='ascii', pattern=/[ -~]*/, exclude_pattern=/\r\n/)
        as LineTerminated
end
```

The interaction The server-side protocol works as follows. Upon establishing a connection, the initial state is called the ServerGreeting state, where the server will then send a greeting to the client in the form of an untagged response. The response can be: (1) an OK, after which the server moves to the NotAuthenticated state; (2) a PREAUTH response, after which the server moves to the Authenticated state; or a BYE after which the server closes the connection.

In the NotAuthenticated state, the server can receive a request from the client to use a specific authentication method. In this case study we only consider a simple LOGIN command with a username and password as the authentication method; when it succeeds, the server moves to the Authenticated state. In the Authenticated state, the server can receive commands that relate to the management of mailboxes. For example after receiving a SELECT command, and if the command is successful, the server moves to another state where commands such as storing and fetching emails are possible. Capturing those interactions in APSL is straightforward. As an illustration, below we show how transitions from the states ServerGreeting and Authenticated are modeled in APSL. The full interaction model, consisting of 7 states and 31 transitions, can be found in [34].

```
actor IMAPServer with
  init state ServerGreeting where
    anytime do send UntaggedOknext NotAuthenticated
          or do send PreAuthGreeting next Authenticated or do send Bye quit
  end
  state Authenticated where
    anytime do send UntaggedOk continue
```

```
    on SelectCmd do next Selecting or do send NoResp continue
    on CreateCmd do send OkResp continue or do send NoResp continue
    on RenameCmd do send OkResp continue or do send NoResp continue
    ...
 end ...
```

4.2   The testing

After the relevant part of IMAP is modeled in APSL, as discussed in Section 4.1, we can proceed to testing. We tested the Courier IMAP server [13] version 4.10.0 as the IUT of the IMAPServer model from Section 4.1. The server was configured to have a mailbox already created, populated with some emails. Since logging-in successfully to the server would problematical for APSL's message generator, we over-specified the username and password fields as we did to MailboxId. In general, a communication channel is needed to connect APSL test engine to an IUT. In this case, a TCP/IP channel is needed, but this is already delivered by APSL itself. No further adapter was needed to facilitate testing. The test engine was configured to keep traversing the IUT up to a certain maximum number of steps T_{max}. As the test strategy, we simply used the default strategy that randomly chooses the next transition to follow if there are more than one possible. This turned out to be sufficient for cover all feasible transitions. Some transitions turned out to be unfeasible in the case study, e.g. the Courier server never actually sends the Bye transition (see the partial definition of IMAPServer actor in Section 4.1) even though this is allowed by the protocol.

Covering all (reachable) messages' fields and instances of enumeration types was more challenging, but still possible, with large enough T_{max}. For example, the command to fetch emails has three variations: to fetch the emails' whole content, or only their envelope, or only their flags. This is represented appropriately within the corresponding field of the fetch-command message by specifying it using an enumeration type. We do not however make the distinction visible at the transition level. That is, in our actor model sending a fetch-command is represented by one transition rather than a choice of three transitions. The latter is expressible in APSL, but would make the actor model overly verbose and hence undesirable. Representing them as a single transition however means that the traversal strategy in Figure 4 also needs to control the message generator so specify which variant of a message type it wants to generate. Such control is currently not possible; this is future work.

No invalid format nor invalid trace error was found during the testing. Manual inspection on the coverage reports of some of the produced traversals did reveal unexpected behavior. When the test engine sends a command to the email server to delete a mailbox named INBOX, the server refuses, which is indicated by sending a NoResp response. This is correct, since IMAP's specification does not allow INBOX to be deleted. However, the investigation revealed that subsequent commands to select INBOX is then rejected (the server replies with a NoResp). While this is allowed by the interaction model that we constructed, this is obviously a bug. With respect to the model, such a bug cannot be revealed by

APSL, nor by any other purely LTS-based testing, since it is neither an invalid format nor an invalid trace error. Refining the model where the special status of INBOX is made explicit as a special message type would reveal the error, but on the other hand would also make the model much more verbose, which is undesirable. A better way to deal with this would be to extend APSL so that transitions can be decorated by post-conditions. This is future work.

5   Related Work

The most used language for describing complex messages in communication protocols is probably Abstract Syntax Notation One (ASN.1) [22]. It was introduced in 1984 and has since then underwent several revisions. It is a joint standard of ISO, IEC, and ITU. ASN.1 is comparable with AMSL without codecs. Translations to the machine representation (the codec part) is expressed in a different language; there are some options, e.g. the Basic Encoding Rules (BER) notation, the XML Encoding Rules (XER) notation, or ITU's own Encoding Control Notation (ECN). However ASN.1 has has grown to be rather large and complex: for example, it has more than 10 different string types, many intended for legacy character encodings. It also has ten expansive standards [21]. These make it more expensive to build an ASN.1-based tool. Eclipse Titan [2] is an example of an ASN.1-enabled testing tool. The tool was originally developed by Ericsson in 2000, but has now been released as an open source project. In Titan, test cases care abstractly written in TTCN-3 [19]. They will be compiled to concrete and executable tests. Messages in the test cases can refer to ASN.1 definitions. Despite its size (over 1.5 million lines of code), Titan is not a model based testing (MBT) tool. In theory, it is possible to integrate e.g. an existing MBT tool with Titan, but we have yet to see such an integration. As far as we know, there is no MBT tool that is ASN.1 enabled out of the box.

In contrast to ASN.1, APSL is a simple language, but it is expressive enough to model complex messages. It already contains MBT tools out of the box. Typical testers only need to use the MBT tool. Its test engine also allows a custom traversal strategy to be hooked, if the default random strategy is not sufficient. Tools builders may wish to modify the test engine itself. The source code of APSL consists of less than 4000 lines of Haskell code, which should be feasible to work with.

In addition to ASN.1, there are indeed more recent protocol description languages, e.g. Google Protocol Buffers [18] and Apache Thrift's interface description language [3]. Both allow for definitions of protocol messages in terms of simple data structures, and code can be generated that operates on the described data. However, these systems use a particular binary encoding method. Therefore, the languages can not be used to describe protocols not originally designed with them. There is also the binpac language [28] which can be used to generate parsers for binary messages. A binpac definition of messages types are compiled to parsers in C++. Pieces of additional C++ code can be mixed in to describe additional parser logic. However, the tool set can only produce parsers.

It does not allow for the construction of messages according to the definition (which is essential for a testing tool).

APSL relies on random generators to generate messages. Generating valid messages turns out to be, perhaps unsurprisingly, easy. The most complex form of constraints on a field is a regular expression. Such an expression can be constructively turned into generators. In comparison, the problem is analogous to generating valid input combinations in automated unit testing of functions or methods. However, functions' inputs are often constrained by a first order formula pre-condition that can be hard for a random based testing tool such as QuickCheck [9] or T3 [31] to solve. If have to deal with a protocol whose messages are constrained in the similar way, we will face the same problem. Fortunately, there are ways to at least mitigate the problem, e.g. by using a theorem prover like Z3 [12] to solve the constraint, or, as in [16,39], by employing a search algorithm [24].

So far we assume the IUT to be a black box. In reality, we may have its source code which we can instrument. In particular, we can insert code to collect runtime information. Even if we do not have the source code, some bytecode can still be instrumented e.g. using tools like ASM [10] or Asil [25]. By collecting such information we can obtained more in-depth information about a test. E.g. if the IUT ever throws an internal exception, and if so, which messages triggered it. We can also use the information to mine common patterns which in turn can be used as anomaly detectors. E.g. it may be revealed that whenever the IUT can either send a or b, it seems to always do the first whenever a certain internal function f returns a 0. An anomaly detector monitors a program at the runtime and will raise a warning if the current behavior violates the behavior pattern it is keyed in. Note that this does not immediately mean that the behavior is erroneous; a human must investigate to decide this. A tool like Daikon can be used to mine from runtime data [14]. More precisely, it mines 'invariants' (properties over program variables that hold on certain points in the program). Alternatively, techniques from data mining such as clustering can also be tried [29].

6   Conclusion and Future Work

We conclude that APSL is able to express real life protocols with complex messages such as IMAP and WebSocket and to provide at least basic automated model-based testing of these protocols. APSL has a test engine that can be parameterized with a traversal strategy. The default random traversal strategy was enough for the email server case study to cover all (feasible) transitions and message fields' variants. However, other protocols may be more challenging, thus requiring more powerful traversal strategies to be written. There are approaches such as combinatorial testing [27], search-based testing [24], or active learning [4] which can be used as a base for such strategies. It requires however further study to know what would work best on which classes of protocols.

The email server case study also shows a draw back of pure LTS model-based testing. The server turned out to contain a real bug which a pure LTS-based approach would be blind without resorting to capturing certain value-level dependencies as separate message types. This can be done, but it will lead to an overly verbose model which is undesired. A better approach would be to extend the LTS-based approach by allowing transitions to be decorated with pre- and post-conditions. This is future work.

APSL's current test engine can only test synchronous protocols. Some protocols, such as WebSocket, are however asynchronous. An actor of such a protocol can buffer incoming messages and is thus not required to respond immediately to an incoming message. Although we did a case study with a WebSocket server [35], we only explored the synchronous behavior of the server by forcing the test engine to wait long enough before it sends the next message to the IUT, and thus we can assume that by that time the server would have consumed all messages in its input buffer. To also test the server's asynchronous behavior the test engine needs to be extended. The challenge of asynchronous testing is that it introduces another aspect of non-determinism [37] that inhibits the test engine's ability to infer the state or states where the IUT might currently be; the number of possibilities can grow exponentially. Although this is a well known phenomenon in asynchronous testing, further study is needed to come up with strategies to keep this explosion manageable.

References

1. Autobahn: Open-source real-time framework for Web, Mobile & Internet of Things, http://autobahn.ws
2. Eclipse Titan, https://projects.eclipse.org/projects/tools.titan
3. Apache Foundation: https://thrift.apache.org/
4. Bauersfeld, S., Vos, T.: A reinforcement learning approach to automated GUI robustness testing. In: Fast Abstracts of the 4th SBSE (2012)
5. Belina, F., Hogrefe, D.: The CCITT-specification and description language SDL. Computer Networks and ISDN Systems 16(4) (1989)
6. BSON: http://bsonspec.org/
7. Budkowski, S., Dembinski, P.: An introduction to estelle: a specification language for distributed systems. Computer Networks and ISDN systems 14(1) (1987)
8. Cabot, J., Gogolla, M.: Object constraint language (OCL): a definitive guide. In: Formal methods for model-driven engineering, pp. 58–90. Springer (2012)
9. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. ACM sigplan notices 46(4) (2011)
10. Consortium, O.: ASM, http://asm.ow2.org/
11. Crispin, M.: RFC 3501: Internet message access protocol–version 4rev1 (2003)
12. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. Springer (2008)
13. Double Precision, I.: Courier IMAP, http://www.courier-mta.org/imap
14. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1) (2007)

15. Fernandez, J.C., Jard, C., Jéron, T., Viho, C.: Using on-the-fly verification techniques for the generation of test suites. In: Int. Conf. on Computer Aided Verification. Springer (1996)
16. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proc. of the 19th ACM SIGSOFT symposium and the 13th European conf. on Foundations of software engineering. pp. 416–419. ACM (2011)
17. Goga, N.: Comparing TorX, autolink, TGV and UIO test algorithms. In: 10th Int. SDL Forum. Springer (2001)
18. Google Developers: https://developers.google.com/protocol-buffers/
19. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Computer Networks 42(3) (2003)
20. Heerink, L., Feenstra, J., Tretmans, J.: Formal test automation: The conference protocol with phact. In: Testing of Communicating Systems. Springer (2000)
21. ITU: Abstract Syntax Notation One (ASN.1) Recommendations, http://www.itu.int/ITU-T/studygroups/com17/languages/
22. ITU: ASN.1 Project, http://www.itu.int/en/ITU-T/asn1
23. Lind, J.: Specifying agent interaction protocols with standard UML. In: International Workshop on Agent-Oriented Software Engineering. Springer (2001)
24. McMinn, P.: Search-based software test data generation: A survey. Software Testing Verification and Reliability 14(2) (2004)
25. Middelkoop, A., Elyasov, A.B., Prasetya, I.S.W.B.: Functional instrumentation of actionscript programs with asil. In: Int. Symposium on Implementation and Application of Functional Languages. pp. 1–16. Springer (2011)
26. Mills, D., Martin, J., Burbank, J., Kasch, W.: RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification. Internet Engineering Task Force (IETF), 2010. IETF
27. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys 43(2) (2011)
28. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: A yacc for writing application protocol parsers. In: Proc. of the 6th ACM SIGCOMM conf. on Internet measurement. pp. 289–300. ACM (2006)
29. Pol, A.: Clustering and dynamic invariant detection (2015)
30. Postel, J., Reynolds, J.: Instructions to rfc authors (1997)
31. Prasetya, I.S.W.B.: T3, a combinator-based random testing tool for Java: Benchmarking. In: Int. Workshop on Future Internet Testing. Springer (2013)
32. Scott, G.D.: Guide for internet standards writers (1998)
33. Smith, G.: The Object-Z specification language, vol. 1. Springer (2012)
34. T. Tervoort: APSL, https://git.science.uu.nl/prase101/apsl
35. Tervoort, T.: A Protocol Specification Language for Testing Implementations. Master's thesis, Utrecht University, Dept. of Information and Comp. Sciences (2016)
36. The TCP/IP Guide: DNS Messaging and Message, Resource Record and Master File Formats, http://www.tcpipguide.com
37. Tretmans, G.J.: A formal approach to conformance testing. Ph.D. thesis, Twente Univ. (1992)
38. Tretmans, J., Brinksma, E.: TorX: Automated model-based testing. In: 1ST European Conf. on Model-Driven Software Engineering (2003)
39. Vos, T., Tonella, P., Prasetya, I.S.W.B., Kruse, P.M., Shehory, O., Bagnato, A., Harman, M.: The FITTEST tool suite for testing future internet applications. In: Int. Workshop on Future Internet Testing. pp. 1–31. Springer (2013)