A Protocol Specification Language for Testing Implementations

Master Thesis

Author: Tom Tervoort
Supervisor: Wishnu Prasetya
Utrecht University
ICA-3470784

Abstract

When developing a system that partakes in a communication protocol, testing whether the implementation conforms to the official specification of that protocol is difficult: the specification may be unclear or ambiguous and creating a simulator that automates tests is time-consuming.

The protocol specification language APSL is proposed, with which one can describe the types of messages used in a protocol, along with a model of the protocol's state machine. It distinguishes itself from similar languages by allowing the generation of message parsers and serializers, and by providing an interaction model useful for conformance testing.

Additionally, an extensible framework is presented that can execute automatic conformance tests in order to determine whether an implementation exhibits the behaviour expressed by an APSL specification.

Contents

1 Introduction

When two or more systems need to communicate with each other, they use a protocol: a previously agreed-upon method of assigning meaning to sequences of bytes that are being transmitted electronically. For example, a PC retrieving the contents of a web page employs a lot of protocols: these take care of matters such as transmitting packages from two machines possibly located on other sides of the world (Internet Protocol), translating domain names to IP-addresses (Domain Name System), establishing a reliable connection between two hosts (Transmission Control Protocol) and indicating how to obtain which resources (Hypertext Transfer Protocol). Protocols are not limited to network applications,though: systems communicating with USB devices or a programs reading files using a particular format also engage in protocol-based communication.

Commonly, a programmer implementing a party of a protocol has no control over the other systems theirs has to communicate with. To prevent incompatibility problems, they strictly have to adhere to a *specification*: a description of what messages to send, what these messages look like and how to respond to messages coming in from outside.

A number of aspects can make the implementation of a protocol, according to a specification, rather complex:

1. Specifications may be unclear or ambiguous, leading to (subtle) differences in interpretation among implementers. Their descriptions may also be incomplete when it comes to corner-cases the writer of the specification had not considered.

2. Protocol implementations usually require parsing and serialization of messages to and from a representation in a programming language data structure. This may be complicated when the data representations within the language and protocol are dissimilar.

3. The system should fail properly when unexpected input (such as messages that do not conform to the specification) is received.

4. Testing is difficult, because it requires one to also find or build a simulator that can send appropriate messages to the system and checks whether it responds correctly. As a protocol grows in complexity, so do its implementations and test suites.

5. When debugging, one may have to capture and decipher network traffic in order to discover what is going wrong.

6. The other systems that will be communicated with may not be trustworthy; meaning that malicious parties may try to find obscure flaws in an implementation and exploit them.

Designing a good protocol is also a difficult task, but once that has been accomplished these complexities make actual implementations of this design expensive and prone to failure or security vulnerabilities. Furthermore, a single protocol design is generally implemented multiple times.

Something that may resolve or alleviate the problems listed above would be a *protocol specification language*: a formal language, readable by both humans and computers, that unambiguously describes what the messages look like, what semantics they have and how exactly an implementation should behave.

When done well, such a language can help implementers in the following ways:

1. It can provide an exact, formal and sound definition, which could ideally only be interpreted in one way.

2. Code for parsers and serializers can be automatically generated from a specification in this language. This will help translating protocol messages from and to data structures in many different programming languages.

3. Code can also be generated to send and receive messages, provide runtime contracts, and to automatically discard or fail on receiving messages that are either ill-formatted or arrive at the wrong moment.

4. Automated test suites can be generated that try to determine whether some implementation indeed conforms to the specification described in the language, and thus to the protocol (assuming the specification is correct).

5. Captured network traffic from and to an implementation can be visualised in a clear and protocol-specific manner when the specification provides the semantics of these raw byte sequences.

6. While one can not perfectly test for security, an automated tester can be tailored to find common security issues such as state machine problems or buffer overflows. Furthermore, a good automated tester may be better at finding obscure edge cases (which often lead to vulnerabilities) than a system that is tested manually.

Besides helping implementers, it could also aid protocol designers:

- A paper specification, intended for humans who are not necessarily implementers, could be automatically generated from the specification, similar to how API documentation is derived from code. A documentation generator could also render nice tables and state or sequence diagrams.
- Tools that help with the analysis or validation of protocol properties require some (high-level) representation of the protocol logic. Perhaps in some cases a specification in this language could be converted to the language used by such a tool.
- Prototyping a new protocol is easier when one can 'program' it in a specification language, especially since much of the implementation code could be automatically generated.

This thesis will present the design of a protocol specification language that is suitable for these applications, along with a compiler and a framework for performing conformance tests.

2 Available Protocol Description Languages

When specifying a protocol in natural language, intended to be read by humans, there are still some standard techniques one could follow in order to make specifications easier to understand by implementers: for example, Request for Comments (RFC) memoranda from the IETF use a specific set of rules for formatting standard descriptions [1]; furthermore, guidelines are available about how to describe an internet standard ([2]).

Even when following a relatively strict description format in natural language, a computer will still not be able to interpret it, and the description may still be ambiguous or incomplete. Numerous formal languages for describing (parts of) protocols are available, tough. These can generally be divided between languages that describe the format of messages that are passed around as part of a protocol, and those that describe the manner in which systems, participating in the protocol, interact.

2.1 Describing Messages

A distinction can be made between *text-based* protocols of which messages correspond to readable text and are defined in terms of character sequences, and *binary* protocols, that are more compact and defined in terms of bit sequences. Different languages exist for these two perspectives.

2.1.1 Textual Messages

Whenever the grammar of protocol messages is *context-free*, one can use a notation technique such as the Backus-Naur Form (BNF) or an extension thereof

to formally describe its syntax [3]. A parser generator such as Yacc [4] could then derive code that can be used to parse it. An extension called the Augmented Backus-Naur Form is specifically aimed for the description of protocol messages [10].

Textual protocol messages are usually based on a particular communication format, such as XML [5]. For protocols that use such formats, there is often also a description language available. For XML, these are the *Document Type Definition* and *XML Schema* languages [6]. Likewise, the *JSON Schema* language formalizes messages using the JSON format [7]. Such schema languages are very useful for describing individual protocol messages.

2.1.2 Binary Messages

ASN.1 When developing a new binary protocol, one could use the specification language ASN.1, which has been used since the 1980's for a wide range of protocols [8]. ASN.1 allows one to describe the data structures that form protocol messages by defining fields with primitive types such as `INTEGER` or `BOOLEAN`, and combining these with sum or product types.

There are multiple standard methods of encoding messages described in ASN.1, using a binary representation or even XML. It is also possible to define custom encoding rules for ASN.1 types using the ECN language, which allows describing protocols in the language that were not originally defined in ASN.1 [11].

Unfortunately ASN.1 is defined by ten expansive standards ([9]), and has grown to be rather large and complex: for example, it has more than 10 different string types, many intended for legacy character encodings. Considering the amount of time the language has been in use, this is not surprising.

Google Protocol Buffers and Thrift Two more modern message description languages are Google Protocol Buffers ([12]) and Apache Thrift's interface description language ([13]). Both allow for definitions of protocol messages in terms of simple data structures, and code can be generated that operates on the described data.

These systems use a particular binary encoding method. Therefore, the languages can not be used to describe protocols not originally designed with them.

binpac The binpac language, described in [14], can be used to define parser generators for binary messages. It is compiled to C++ code, and pieces of C++ can even be mixed in to describe additional parser logic.

This language is solely intended for the generation of parsers, it does not allow for the construction of messages according to the definition.

2.2 Describing Interactions

Messages are just one part of a protocol: equally importasnt are the *interactions* between systems that follow a protocol; i.e. under what condition can what system send, or expect to receive, what message.

Different models exist that formalize these possible interactions, for example:

- *Petri nets* are state-transition systems (similar to finite state machines, but with some different properties: such as no requirement to have a finite amount of states or transitions). They consist of 'places' and 'transitions'; a place may contain a number of 'input tokens' and, when sufficient are present in a place connected to some transition, they can be consumed and be converted to output tokens, which are send to other end of the transition [30].

- The *communicating finite state machine* model is an extension of the finite state machine model, in which transitions are labelled with a type of message they should either receive or will transmit [29].

- The *labelled transition system* model is similar to that communicating finite state machines: however, it does not model all communicating systems at the same time but rather a single one around which *observable* (the reception and transmission of messages) and *internal* events can trigger state transitions. It is described in more detail in Section 3.3.

There are many tools available that operate on descriptions of such models, and allow one to prove that a certain protocol has certain properties. The SPIN model checker, for example, allows for automatic verification of certain properties (expressed in temporal logic) of a high-level protocol description (using a special description language that is based on non-deterministic state machines) [31].

Such tools are generally aimed at the *designers* of protocols. I am, however, looking for a language that can aid *implementers* of a protocol. Properties of the protocol itself are not relevant in this context: instead, one wants to test whether a certain implementation correctly matches a certain protocol description, regardless of what that protocol is actually supposed to do.

2.2.1 SDL

A formal language that could be used for this purpose is the Specification and Description Language (SDL) [15]. This language was originally designed in 1976 and has both a visual and textual representation. It is a very broad and large language that can be used to describe many kinds of distributed and reactive systems. The structure and communications of those systems can be described using a model based on *extended finite state machines*. Code and UML diagrams can be generated from these descriptions and interoperation with ASN.1 is possible.

Unfortunately, SDL is so expansive and complex that the creation of tools operating on its models is very difficult. In fact, currently none of the (usually commercial) SDL tools supports the entirety of the language, instead they all operate on specific subsets [16].

2.2.2 Estelle

Estelle is another formal description technique for protocol specification, it was defined in the 1980's and based on the Pascal programming language. It models a protocol as a set of communicating finite state machines [17].

An Estelle specification (or at least a subset thereof) can be used to semi-automatically generate parts of a protocol implementation [18] and has been successfully applied during the design of a protocol for military mobile combat radios [19].

However, Estelle models protocol messages using Pascal types, but can not be combined with message descriptor languages such as those described in Section 2.1. Therefore an Estelle model is not enough to test conformance of a protocol implementation: one still has to manually build and parse the messages.

3 Protocol Implementation Testing

Given a description of protocol messages and possible interactions, it becomes possible to generate a *test suite* that tries to find out whether some implementation on some system conforms to the protocol in question. Note that *testing* does not equal *verification*: while a test suite may be able to identify the existence of an error, it can not prove a given property holds in all cases. It can however, significantly increase confidence in the level of correctness of an implementation.

Protocol testing is usually a form of *black-box testing*: the tester only has access to a particular interface to a system (e.g. a communication channel through which messages can be send and received), but can not 'peek inside' of the tested system and observe its internal state. [20]

3.1 A Protocol Participant as a Finite State Machine

Each system that participates in a protocol could be modelled as a *finite state machine* (FSM): a machine that is always in one of a finite amount of *states* and can *transition* from one state to another when prompted to by an (external) *event*.

When using the *Mealy* definition of FSMs [21], such a machine is a quintuple $(I, O, S, \delta, \gamma)$, where:

- I and O are finite, non-empty sets of respectively input and output symbols. In this context: types of messages that can be received by or send from the modelled machine.

- S is a finite, non-empty set of possible states.
- $\delta : S \times I \to S$ is a a state transition function: it determines to which new state to move upon receiving a particular input, depending on the current state.
- $\gamma : S \times I \to O$ is an output function: it tells which output to generate when receiving a certain input in a certain state. δ and γ together describe the *transitions* of an FSM.

Conformance testing of an implementation to a specification can be accomplished by deriving an FSM M_{spec} from the protocol description, and by also perceiving the physical system to be tested as an FSM M_{real}. We can now test conformance by testing whether M_{spec} and M_{real} are *equivalent* [21].

In a black-box testing scenario, one can not see the internals of M_{real}. But it is possible to send messages to it and observe its outputs, i.e. its δ and γ functions can be executed. By comparing the outputs M_{spec} generates to those of M_{real}, given the same inputs, a case could be found where they do not match; in that case the two machines are not equivalent and thus the implementation does not conform to the specification. Note that outputs can only be compared when one knows what state the system currently is supposed to be in.

When it is not known what the current or initial state of the machine represented by M_{real} is, one can try to use *state discovery* techniques (which also involve sending inputs and observing outputs) in order to find out [21].

Note that in order to be able to view a real system as an FSM in this manner, one usually assumes its state machine graph is *strongly connected* (i.e. every one state is eventually reachable from another, otherwise it would not be possible to reach all states while testing). Furthermore, it is also assumed the machine is *reduced* (no two separate states are equivalent, meaning they can not be distinguished by comparing input/output sequences), and that a system's machine does not change during the experiment [21].

Under these assumptions, one can try to generate tests by trying to *cover* all the states and transitions the test subject's FSM is supposed to have. [22] describes a randomized algorithm that provides the sequences of inputs one has to send to a machine in order to achieve this.

3.2 Partially Specified and Non-deterministic FSMs

The previous section assumes a complete FSM can be derived from a protocol model, but such a model may not describe every possible interaction: particularly when an 'unexpected' message comes in while at a certain state. There are techniques for testing a partially specified machine, but in general it is probably simpler (and appropriate) to transform it into a completely specified FSM by making sure each state has a transition that is activated when receiving a particular 'undefined' message, and then by transforming each input not accounted for into such a message [20]. The specification language may allow the

behaviour upon receiving an undefined message to be described, or could insert transitions to some kind of 'error state' by default.

Protocol models may also contain *non-determinism*: multiple transitions could be possible from the same state, given the same input; which transition is chosen may be random or depend on outside factors not present in the model. This means that the same test may need to be executed multiple times, until all transitions have been encountered. This may not always occur, though, and therefore one has to make some assumptions on when an adequatly large section of the model has been tested, even if not every transition was visited. [20]

3.3 Labelled Transition Systems

A concrete model, based on the finite state machine representation of protocols described in Section 3.1, that can be used during conformance tests is that of *labelled transition systems* (LTS). Such a system consists of the following [38]:

- A non-empty set of states S.
- A set of *observable actions* L. In this context, these actions consist of messages that are perceived to have been sent to the system (input actions), and messages that the system itself produces (output actions). However, they are treated in the same manner by the model.
- A set of transitions T, each being a triple $(s_1 \in S, a \in (L \cup \{\tau\}), s_2 \in S)$. They indicate that the system should move from state s_1 to s_2 once the action a occurs. The symbol τ represents an *internal action*; these are any events that are not observed but can still trigger a transition.
- An initial state $s_0 \in S$. The system starts in this.

Using a labelled transition system, one can predict the state a system is supposed to be in (assuming it is correctly reflected by the model), just by examining the trace of observable actions. However, when a transition is non-deterministic (there exist two transitions (s_1, a, s_2) and (s_1, a, s_3) for distinct states s_2 and s_3), a system could potentially be in more than one state when a is observed while in s_1; this is also the case when a transition exists from s_1 with action τ, since it can not be directly observed whether the internal action has taken place or not.

When, for example, the two possible states of a system are s_2 and s_3, the uncertainty could be cleared up by observing a following action: if a transition with this action exists for s_2 but not s_3, it becomes clear that the system must have previously been in s_2.

3.3.1 Deriving Conformance Tests

Tretmans describes a method for deriving *sound* conformance test cases for a protocol participant from a labelled transition system [38].

A system can be shown to be nonconforming when a sequence of observable actions is perceived that should be impossible according to the model. For example, given the LTS $(S = \{x, y\}, L = \{a, b\}, T = \{(x, a, y), (y, b, x)\}, s_0 = x)$, the *trace abb* is invalid: after ab, the system should be in state x, but from there no transition exists for observable action b.

A tester can influence the observable actions that happen by sending a message to the system, which results in an *input action* to occur in it; however, the tester can not force *output actions* to take place but only observe them. Therefore, in order for tests to be sound, the tester should never send a message that would result in the action trace becoming invalid; at the same time, it should detect when an output action generated by the system renders the sequence invalid, in which case an error has been successfully detected.

By following these rules a tester is sound but not necessarily complete or even useful: a tester that only observes but does not produce any messages is an example of a sound test mechanism that is not likely to find any problems. Therefore, one needs a strategy that determines when the tester will trigger which input action (i.e. when it will send what message type). Section 5.2.1 describes one such strategy.

3.3.2 Asynchronous Testing

In Section 3.3.1, it is implicitly assumed that a *tester* (the automated tool that performs the test) can completely decide in what order the system will process input actions (the messages the tester sends to the system) as opposed to output actions (the messages the system sends to the tester). However, this is not always realistic: consider the case where the system and tester respectively transmit a message x and y at roughly the same time (x is transmitted before y is processed or vice-versa); now, after both parties receive each other's messages, the tester will think that the action trace so far is yx while the system has actually processed the sequence xy. Due to this inconsistency the tester may no longer be sound and generate false positives.

This problem does not occur in a *synchronous* communication context, in which systems send messages one-by-one but could never decide to do so at the same time. It is also not a problem when no two transitions (a, x, b) and (a, y, c) exist, where x is an input action and y is an output action, since no ambiguity can arise.

In an *asynchronous* context however, in which the model may allow for simultaneous message generation, it is more difficult to derive a sound tester. A method to achieve this is described in [38], and it boils down to the tester needing to maintain a separate input and output trace. The actual action trace is some combination of these, in which input and outputs have the same order with regards to themselves but of which the ordering regarding each other is unclear to the tester. An error can only be detected when traces are observed for which all possible combinations will lead to an invalid action sequence; furthermore, the tester may not append to the input trace (i.e. send messages) in a way that could allow for any invalid sequence when combined with the current output

trace.

3.4 Fuzzing

The previous sections deal with *state machine testing*, which try to explore all protocol states and uncover invalid transitions. However, even when a system correctly transfers from state to state, messages that are large, invalid, 'strange', or contain particular values (i.e. an integer near to its maximal boundary) may also result in incorrect behaviour. These problems are often caused by buffer or integer overflow issues.

Fuzzing is a technique designed to uncover these kinds of issues. Basically this involves the generation of a large amount of *randomly generated* messages (simply random bitstreams or randomly chosen values that fit particular criteria), that are send to the system. Then one tries to observe whether it still behaves correctly [23].

Fuzzing can be a useful and easy way to test software: for example, the Quickcheck tool ([25]) can be provided with a means to randomly generate values of a certain type (methods of generating standard types such as integers or strings are already build in), and will then use this in order to fuzz the inputs of a certain predicate (i.e. the test case). When the predicate does not hold, the tool first tries the 'simplify' the inputs to a form that is less complex but still causes failure; then it reports those to the user.

The Peach framework ([24]) is an example of a tool that applies this technique to network protocols: given a programmatic definition of protocol messages, it tries to generate many random messages according to this definition (and some which intentionally do not match it). Then it fires those at a system while observing whether it crashes or exhibits incorrect behaviour.

When combined with state machine testing and a message specification, fuzzing could be a simple and effective technique with which to enhance a protocol's test suite. An interesting example of a fuzzing tool that combines these techniques is SMACK ([26]), which has been specifically designed for testing implementations of the TLS protocol. It has already succeeded in finding two major security issues in this part of critical infrastructure, that were present in multiple popular implementations including OpenSSL.

4 The APSL Language

APSL (an uncreative acronym meaning *A Protocol Specification Language*) is the language I designed that attempts to solve the problem described in Section 1: i.e. to allow one to transcribe a protocol in such a way that automated conformance tests and other implementation aids can be automatically derived from it. APSL has the following important properties:

1. APSL is actually split in two sub-languages: AMSL and AISL; the first allows the description of message structure and reusable components that

can be used within messages, the latter can be used to specify a *state machine* that details which messages can be sent or received under what circumstances.

2. If desired, the two sub-languages can be used separately and the functionality of one of the two languages could be replaced by something else.

3. Generally, it should not be too difficult to translate a formal English-language description of a binary message protocol to an APSL specification. With *binary message* protocol I mean that the messages consist of compact bitstrings rather than human-readable text constructed according to a complex grammar (such as XML). There are also binary protocols which APSL can not adequately express (see 5.1.2), but based on the experience of successfully specifying multiple popular protocols, I do believe it to be quite widely applicable.

4. A compiler for the language has been created, along with an extensible framework for conformance test derivation. Furthermore, the framework makes it easy to develop other applications for the language. See Section 4.3.

5. A level of abstraction has been chosen that allows the automatic derivation of message parsers and generators which are capable of decoding messages transmitted by a real implementation and can provide valid responses. However, APSL is not more low-level than necessary: it is not a programming language in which all protocol logic can be expressed.

6. When an inconsistency is found between the behaviour of an implementation and an APSL description, either of them must be incorrect: this means that, given a correct APSL specification, testers looking for errors should not yield false positives, but may miss problems; i.e. the specification is *sound* but not necessarily *complete*.

The AMSL and AISL languages have similar syntax, and AMSL modules can be imported into AISL specifications. An APSL specification consists of one or more AMSL modules and one or more AISL modules that depend on them.

The syntax of both languages is not whitespace-sensitive, and uses keywords rather than symbols to structure statements.

4.1 The Message Description Language: AMSL

An AMSL (*A Message Specification Language*) file contains a single *module*. This module may import definitions from other modules and define the structure of zero or more *messages*.

A grammar and precise description of the language can be found among the source code of the tool (see Section 9).

4.1.1 Fields

Messages are compositions of *fields*, which comprise the core building blocks of an AMSL specification. Each field has a name and an *abstract data type*, which defines how the contents of this field should be interpreted. Examples of AMSL's data types are the self-explanatory `Integer` and `Text`.

Unlike the specification languages Protocol Buffer and Thrift (Section 2) however, a field with a certain data type does not need to be encoded in a single specific way; aside from the type, a field is assigned a *data codec*, which is the mechanism of converting a value of this data type to and from a bit string. For example:

```
feature_enabled is Bool as BoolBits(truth_string=b1)
```

The field `feature_enabled` has the type `Bool`, which means it can either have the value `true` or `false`. `BoolBits` is a codec that indicates how either value should be serialized; the parenthesized part following it is a set of *codec arguments* that provide further encoding instructions. In the case of `BoolBits`, the `truth_string` argument provides a constant bitstring (in this case the *binary literal* `b1`; i.e. a single 1-bit) that should be used to represent the value `true`; `false` is expressed by any other bitstring of equal length (in this case, one 0-bit).

The type of a field can also be given arguments. These allow the restriction of the set of values that are permissible within a field: after the decoder has used the given codec to transform a bit sequence into an instance of the type, it is verified whether the predicate expressed by these arguments holds for this value; if not, the input is rejected. This can be seen as a form of *refinement typing*. Some examples:

```
five_chars  is Text(count=5, pattern=/[a-zA-Z]*/)
                as FixedCountText(encoding='utf-8')
some_number is Integer(min=0, max=2^16-1)
                as BigEndian(length=16, signed=false)
some_bytes  is Binary(max_length=256)
                as LengthPrefixBinary(
                    length_factor=8,
                    length_codec=
                     BigEndian(length=8, signed=false))
```

It should be noted that the codec and codec arguments do not affect the set of values that can be stored in a field. However, the type and type arguments do influence which codecs can and can not be used: for example, `FixedCountText` can only be assigned to `Text` fields that set an explicit value for `count`.

AMSL defines a (dynamic and soft-typed) *expression language* that can be used to assign values to arguments. Expressions can be primitive values such as `false` or `256`, but they can also contain conditionals, regular expressions, and logic and arithmetic operators. An example of a more complicated expression is `x + 1 if x > y and y >= z else y % (z // 2)`.

4.1.2 Basic Types and Codecs

The types and codecs that have been demonstrated so far are examples of *basic types* and *basic codecs*. These are built-in and what they mean is part of the AMSL language specification. A list of them, along with a description of their arguments, can be found in the file `doc/basic-types-codecs.rst`.

Users can also introduce new types by using the constructions described in the following sections.

4.1.3 Records

A *record* is a kind of new type a user can declare and then reuse: it has a name and contains a composition of multiple fields, for example:

```
record ArchivedFile with
    indicator   is Text(value="FILE")
                    as FixedCountText(encoding='ascii')
    filename    is Text(max_count=100)
                    as CountPrefixText(count_codec=
                        BigEndian(length=8, signed=false))
    permissions is Binary(length=9)
                    as FixedLengthBinary
    size        is Integer(min=0, max=2^64-1)
                    as BigEndian(length=64, signed=false)
    create_year is Integer(min=1970, max=2100)
                    as TextInteger(base=10, count=4,
                                   encoding='ascii')
    file_data   is Binary(length=size*8)
                    as FixedLengthBinary
end
```

The name of a previous field can be used within expressions in the arguments of the following fields; these are automatically filled in while parsing the record. In the above example, the value of the `size` field is reused in order to determine the length of the `file_data` field; it is multiplied by eight because the `length` arguments expects an amount of bits while the size field of this fictional protocol stores amounts of bytes.

Allowing record field to be dependent on each other enables the expression of many common patterns, particularly variable-length fields.

Since a record declaration introduces a new type, this type can be reused within other records of the same module (or within another module that imports the record definition). Furthermore, records can also be *parametrized*: record parameters are the possible type arguments that can be given to it; they can be used within expressions in its fields. An example:

```
record MessageFrame(sender, payload_size) with
    opcode  is Integer(value=0 if sender == 'client' else 1)
```

```
                as BigEndian(length=64, signed=false)
    payload is Binary(length=payload_size * 8)
                as FixedLengthBinary
end

record ServerFrame with
    frame is MessageFrame(sender='server',
                          payload_size=32)
end
record ClientFrame(extra_data_length) with
    frame is MessageFrame(sender='client',
                          payload_size=64 + extra_data_length)
end
```

Note that the codec of a record field can be omitted, in which case the default `RecordCodec` is used. This codec simply assumes the binary representations of the fields simply follow each other directly and decodes them one by one. It is always possible to do this, because for AMSL codecs it is never allowed to leave ambiguous where the bit sequence associated with a field ends.

4.1.4 Unions

Sometimes, a protocol may require that a message is structurally different, depending on the value of a field. In such a case, a union can be defined. Just like a record, a union is declared as a sequence of fields; however, a union value can only contain exactly one of these fields. An example:

```
union PathLookupResult tagged Text of
    "FILE" tags file      is ArchivedFile
    "DIR"  tags dir       is ArchivedDirectory
    "LINK" tags link      is Path
    "NONE" tags not_found is Text(value="NOT FOUND")
                              as FixedCountText(encoding='ascii')
end
```

Each field is associated with a *tag value*. The tag in question is an argument of this union type, and it determines which of these fields is actually used. The type of tag is also declared, and in this case a textual tag is used.

When using a particular codec named `TagPrefixUnion`, the specific value of the tag does not need to be provided as an argument: instead, it is always placed right in front of the union and encoded along with it. However, one does not to provide a codec for this embedded tag value. An example:

```
lookup_result is PathLookupResult
                  as TagPrefixUnion(tag_codec=
                      TerminatedText(encoding='ascii',
                                     terminator="\0"))
```

4.1.5 Enumerations

Some fields can have a small fixed number of values. An *enum type* can be constructed for these situations: it consists of a number of names which each represent a fresh value. An enum type is *backed* by a previously-defined type, of which it inherits the possible codecs:

```
enum DNSRecordType of Integer with
    a     as 1
    aaaa  as 28
    ns    as 2
    cname as 3
    soa   as 6
    ptr   as 12
    mx    as 15
    txt   as 16
end
```

Enumerations are often useful as tag types of unions.

4.1.6 Aliases and Default Codecs

It can be tedious to constantly have to type the same type or codec arguments for multiple similar fields, especially since most protocols predefine a small number of data types that are always encoded in the same manner.

In order to make AMSL specifications more comfortable to write and read, and to reflect protocol-specific versions of data types, it is possible to define *aliases* of existing types and codecs:

```
type  Int64      is Integer(min=-2^63, max=2^63-1)
codec Int64Codec is BigEndian(length=64, signed=true)
```

Note that, besides providing an alternative name for a type, one can also set *default arguments* on an alias type: now, whenever the `Int64` type is used it will be translated to `Integer` and have the `min` and `max` arguments filled in when those are not yet present.

One can also associate *default codecs* with a particular type (including type aliases). When using this type in a field but when omitting the codec, this default is used:

```
default Int64 as Int64Codec
```

4.1.7 Extensions

In case the user wants to introduce a type or codec, but is unable to properly express it in terms of records, unions, enums and basic types, they can declare an *extension*. For example:

```
extension codec GZippedBytes(compress_factor) of Binary
```

Now `GZippedBytes` can be used anywhere within the AMSL module, and it can be given an argument called `compress_factor`.

When actually using a module that declares extensions for an application such as automated testing, the user will have to provide *plug-in code* that can perform certain kinds of operations on data using this extension type. When defining an extension codec, for example, the plug-in will need to interpret its arguments, and be able to perform appropriate encode and decode operations.

At the time of writing, this testing framework does not yet support plug-in code, meaning that it can not do anything useful with modules containing extensions, other than compiling and validating them.

4.1.8 Codec Stacks

While experimenting with the language, a limitation was encountered concerning *redundant length fields*: when a protocol specifies a component that can be modelled as a record, but requires it to be preceded by its binary length (even though this length could be inferred from the record contents), it becomes difficult to express in AMSL.

There is no option for fixing the encoded length of a record, and adding that would be hard and violate the separation between abstract data types and encodings. However, a `LengthPrefixBinary` codec does exist, which encodes values of the `Binary` type and automatically prepends its length; it could not be applied to record types, though, so therefore the `DoubleCodec` was introduced:

```
record_with_length is SomeRecord as DoubleCodec(
                        field=RecordCodec,
                        additional=LengthPrefixBinary(
                                    length_codec=Int64Codec,
                                    length_multiplier=8))
```

`DoubleCodec` can be used with any type, its `field` argument must be applicable to that type, and `additional` must be some codec that can be used with the `Binary` type. Encoding works by first applying the `field` codec and then encoding the resulting bits again with the `additional` codec. Decoding does the same in reverse.

Codec stacking has other applications, particularly when combined with extensions. One could, for example, use this to represent records which are compressed, encrypted or have an appended digest. For example:

```
extension Codec Compressed(algo) of Binary
extension Codec WithChecksum(algo, position) of Binary

codec CompressedAndCheckedRecord is DoubleCodec(
 field=RecordCodec,
 additional=DoubleCodec(
```

```
field=Compressed(algo='deflate'),
additional=WithChecksum(algo='crc32', position='after')))
```

4.1.9 Messages

A message declaration is identical to a record declaration, except that the keyword `record` is replaced by `message` and that it is not allowed for messages to have parameters.

The names of declared messages are the only items which are exposed to the AISL language, and a separate one should be declared for each kind of protocol message one would want to distinguish.

Many protocols have different types of messages which are structured very similarly. To prevent repeating oneself, the structure could be expressed by a single parametrized record type, and the messages simply contain an instance of this record. For example:

```
record Frame(opcode, has_payload) with ... end

message Discover with
    frame is Frame(opcode='DISC', has_payload=false)
end
message AnnounceExistence with
    frame is Frame(opcode='ANN', has_payload=false)
end
message Notification with
    frame is Frame(opcode='NOT', has_payload=true)
end
```

4.2 The State Modelling Language: AISL

Whereas AMSL can be used to describe what the structure of messages looks like and how message fields are constrained, AISL (*An Interactions Specification Language*) allows one to specify under what circumstances a system may receive and/or send particular messages and how those affect the state of the protocol.

As shown in Section 2.2, various types of models and modelling languages exist that can be used to model a protocol as a state machine and describe how the transmission and reception of messages affects transitions between protocol states. I have chosen to base AISL on the model of *Labelled Transition Systems*: this model is described in Section 3.3, which also outlines how this model can be useful for the derivation of conformance tests.

An AISL file contains a single module which can *import* specific or all messages from multiple AMSL modules; these messages are the only aspect of the AMSL language (aside from a similar syntax) that are reused in AISL. Furthermore, the contents of messages are not exposed: the AISL module can only refer to the names of whole messages, but not to their fields.

In fact, the AISL module only needs to be provided by a set of message labels and, when testing, with a method of transmitting or receiving messages with

a particular label. One could still use it in combination with another message description language (such as one more suitable for text-based protocols); or with any piece of software capable of somehow sending a message by label, and giving a notification when a message with a certain label is received.

The AISL language is considerably more simple than the AMSL language, as it is has a much lower amount of language constructs and no built-in "standard library" such as AMSL's collection of basic types. This smaller amount of complexity is evidenced by the number of non-whitespace lines of the grammar specifications of both languages (both using the same parser generator language): 94 for AMSL and 29 for AISL.

4.2.1 Actors

Besides message import statements, an AISL module contains one or more *actors*. Each of these represent an entity that partakes in a protocol. Actors sharing a module have access to the same message namespace, but the language does not offer another method through which components defined in one language can be reused in another.

Conceptionally, a module is supposed to represent a protocol and each actor is supposed to represent a type of entity that can partake in it. However, the language does not forbid one from spreading related actors over multiple modules, or putting actors that are used in different protocols within the same module: actors are the core components of the AISL language and modules are merely collections of them.

An actor represents the state machine of a protocol entity: it describes when it may send messages and how it reacts to incoming ones. A simple example:

```
actor CoffeeMachine with
    init state Idle where
        on EnoughCoinsInserted do
        next HasCash

        on CoffeeTypeChosen do
            send NeedMoreMoney
        continue

        on ShutdownButtonPressed do
            send Goodbye
        quit
    end

    state HasCash where
        on CoffeeTypeChosen do
            send PleaseWait
        next Brewing
    end
```

```
    state Brewing where
        on CoffeeDone where
            send CoffeeAlert
            send Coffee
        next Idle

        on CoffeeTypeChosen do
            send PleaseBePatient
        continue
    end
end
```

An actor has multiple *states*, of which exactly one must be marked as the `init` state (the one the system is in when starting operation). Each state contains zero or more *events*, which have three components:

- The label of a message that will trigger the event upon reception.
- Zero or more `send` expressions, which are all coupled with the label of a message that is supposed to be sent in this occasion. The messages should be transmitted in exactly the order in which they are listed.
- A `next` expression which indicates the next state the system should traverse to. The next message received, even if it directly follows the current one, will trigger an event of the next state. The expression `continue` indicates the system should remain in the same state, and is equivalent to using `next` followed by the name of the current state.

When ending an event with a `quit` expression, the system traverses to an implicit *exit state*; a state that contains no events and can therefore not be recovered from by receiving messages. This models system or session termination.

4.2.2 Invisible Triggers and Non-determinism

If a system could only react upon receiving a message, communicating would be impossible: one party has to be the first to send a message in order to start the protocol. Therefore, the language also allows for events that can occur at any time, without being triggered by an incoming message: to specify such an event, use the `anytime` keyword instead of `on SomeMessageLabel` for the event trigger description.

`anytime`-triggers are considered *invisible*, because they are activated by an outside condition such as user interaction, a timer or an internal error. Just like regular events, it is not required to send any messages; this means anytime-events can also cause invisible state transitions.

When and whether such an invisible trigger activates is non-deterministic; however, systems may also non-deterministically decide how to respond to a

certain message: this can, for example, come from an internal process within the system that is not expressed in the protocol. Non-determinism within an event can be expressed by using the `or`-operator, such as in this example:

```
actor MailBox with
    init state Ready where
        on SendMail do
            send OkMailSent
        continue
        or do
            send FailInvalidAddress
        continue
        or do
            send FailTransmitError
        continue

        on FetchMail do
        next FetchingMail

        anytime do
            send WarningMailboxFull
        continue
    end

    state FetchingMail where
        anytime do
            send NewMail
        continue
        or do
            send NoMoreNewMessages
        next Ready
    end
end
```

4.2.3 Invalid messages

A protocol may define specific behaviour for when it receives a message it does not understand or which should not have been transmitted to it in the current state. This can be expressed using the final language construct: `otherwise` events. These are events that are triggered upon receiving a message that is invalid, or for which the label does not match any other event trigger in the current state.

When not adding an `otherwise` event to a state, the action which the system should undertake when getting an is invalid message is not expressed. In this case, one could assume some default error-handling behaviour for these situations, such as `otherwise do quit`, or treat these as transitions to some

implicit error state.

An example using this construction:

```
actor SomeServer with
    init state Active where
        on Request do
            send Response
        continue

        otherwise do
            send InvalidRequestError
        continue
    end
end
```

4.2.4 Deriving an LTS from an Actor

A labelled transition system, as described in Section 3.3, can be derived from an actor in the following manner:

- First, treat each `otherwise` event as a receive event that is replicated for every imported message label for which no receive event is specified in that state.
- Let the set of *LTS states* contain an entry for each *AISL state* in the actor, but also add an *intermediate* state corresponding to each send action of every event of every state. Add one *exit state* that can be accessed with a `quit` expression.
- The *initial state* is the one corresponding to the AISL state marked with the `init` keyword.
- Let there be an *observable action* for the *reception* of each message label, and for the *sending* of each label.
- For each event, include an *LTS transition* from the AISL (i.e. non-intermediate) state to the first intermediate state of the event (or the next AISL state, in case there is none). If the event has an `on <label>` trigger, let the transition action be the receive action of this label; for an `anytime` trigger, let the action be the unobservable internal action τ. When an `or` operator is used, add transitions for all branches.
- Also add a transition between every intermediate state and its follow-up, that is associated with the send action of the label being sent in this state.

By treating the sending and receiving of messages as observable actions, and by using the *internal action* for `anytime` events, one can apply testing techniques for LTS's to AISL specifications: a testing tool that acts as a protocol simulator

is capable of recording a trace of observable actions, because it knows messages it receives must have been sent by the tested system, and it can observe the messages the system receives because it sends them itself.

4.3 The Compiler and Framework

While the primary purpose of the APSL language is conformance testing, it may also have other useful applications such as those listed in Section 1. Therefore, the compiler of APSL modules has been designed as a library, that transforms them into representations within the Haskell programming language, and also associates APSL types with Haskell types. This should make it relatively simple to write an application that can make decisions based on an APSL specification. The testing framework is one such an application.

4.3.1 Parsing AMSL and AISL

The two sub-languages, AMSL and AISL, both have a distinct grammar (although with a similar syntactic style). The grammars are defined using a *labelled Backus-Naur Form* notation: specifically, the dialect used by the BNFC tool [36]. Haskell code containing a parser and abstract syntax tree representation, has been automatically generated from these grammars using this tool.

In the case of the AMSL language, an abstract syntax tree is transformed to an abstract representation called a `CompiledUnit`:

```
data CompiledUnit = CompiledUnit {
 requiredDependencies :: [FilePath],
 compiledModule       :: [Module] -> Either ASTCompileError Module
}
```

Such a compiled unit consists of a list of paths corresponding the import statements that may have been part of the AMSL module, and a function that may produce an eventual `Module` object (see Section 4.1) once compiled versions of the dependencies have been provided. A `compile` function is provided that can be given a single AMSL file to transform into a `Module`; it will also recursively handle its dependencies when required.

Compilation of AMSL is complicated by the fact that declarations within the same module (such as record definitions and type aliases) may appear in any order but can also refer to each other. In order to cope with this, components of the module are compiled within the following monad:

```
data Dependant a
 = Final a
 | Partial TypeName (Declaration -> Dependant a)
 | RequiresDefaultCodec TypeName (DefaultCodec -> Dependant a)
 | Error ASTCompileError
```

A `Dependant` value may depend on the compiled value of another declaration (or a default codec definition, which is treated separately). The compilation process starts by building a table of `Dependant` declarations, which is effectively a dependency graph; then it is checked that this graph does not contain cycles (recursive structures are not allowed) and does not depend on unknown `TypeName`'s; finally, the dependencies are resolved by computing the `Dependant` values in the correct order.

The AISL compiler is somewhat similar, but considerably more simple due to the fact that AISL structures can only refer to AMSL messages but not to each other.

4.3.2 Representing AMSL Modules and Types

An AMSL `Module` data type is defined as follows:

```
-- | Represents the relevant definitions from a module.
data Module = Module {
 moduleName  :: String,
 messages    :: [Message],
 types       :: [DefinedType],
 codecs      :: [DefinedCodec]
}
-- | An instance of a particular message, of which all of its
--   fields have been given a particular value.
data MessageInstance where
 MessageInstance :: forall a. TypeName -> Record a -> a
                              -> MessageInstance

-- | User-defined (non-message) records, unions, aliases, and
--   enums.
data DefinedType where
 DefinedType :: forall a. UserType a -> DefinedType

-- | Codec aliases.
data DefinedCodec where
 DefinedCodec :: UserCodec -> DefinedCodec
```

Basically, a module contains a number of named types and codecs. The type representations `Record` and `UserType` are existentially quantified over their type parameter, which is an appropriate Haskell type that can encode all members of the represented AMSL type. Both are instances of a more general `Type` construct:

```
data Type a where
 BasicType :: BasicType a -> Type a
 UserType  :: UserType  a -> Type a
 ExtType   :: ExtTypeMeta -> Type ExtData
```

```
data BasicType :: * -> * where
 Binary     :: BasicType BitString
 Bool       :: BasicType Bool
 Integer    :: BasicType Integer
 Text       :: BasicType String
 Optional   :: BasicType (ContainerValue Maybe)
 List       :: BasicType (ContainerValue [])

data ContainerValue :: (* -> *) -> * where
 ContainerValue :: forall f a. Type a -> f a -> ContainerValue f

data UserType a where
 Record      :: TypeName -> [ParamName] -> Record a -> UserType a
 Union       :: forall a tag. TypeName -> [ParamName]
                 -> TagType tag -> Union tag a -> UserType a
 Enumeration :: TypeName -> TagType a -> Enumeration a -> UserType a
 TypeAlias   :: TypeName -> Type a -> Arguments -> UserType a
```

A `Type` and its contents are GADT's: this allows a user to pattern match upon them and obtain evidence of what concrete Haskell type `a` is (even when it is quantified over). The basic types use appropriate Haskell types that correspond to them, and the container types contain an additional `Type` object that provides the type of their contents.

Type aliases simply contain another type with the same representation; the other user-defined types wrap the following GADT's:

```
    data Record :: * -> * where
        Empty :: Record ()
        Field :: forall t c r. FieldName -> FieldTypeCodec t
                  -> Record r -> Record (t, r)

    data Union tag :: * -> * where
        None   :: Union tag Void
        Option :: forall tag t c r. tag -> Maybe FieldName
                   -> FieldTypeCodec t -> Union tag r
                   -> Union tag (Either t r)

    data Enumeration a where
        EnumValues :: [(FieldName, a)] -> Enumeration a

    data FieldTypeCodec t = FieldTypeCodec {
        fieldType :: Type t,
        fieldTypeArgs :: Arguments,
        codecType :: Codec,
```

```
    codecTypeArgs :: Arguments
}
```

Records and unions respectively use Haskell tuples and the `Either` type to combine their elements; additionally, unions define a *tag value* for each option. The `Enumeration` type has a fixed set of predefined `a` values it may contain.

Records fields each have a `Type` and a `Codec`, the latter being a simple type representing either one of the build-in codecs, or a user-specified alias. They are also associated with `Arguments` data types, which are described in the next Section.

4.3.3 Arguments and Expressions

Arguments are a sequence of named *expressions*, or type or codec names with their own respective arguments. The data type is defined as follows:

```
-- Note: these definitions have been simplified a bit
-- compared to the actual implementation.
newtype Arguments = Arguments [(ParamName, Argument)]
data Argument
    = ExpArg Expression
    | forall a. TypeExpArg (Type a) Arguments
    | CodecExpArg Codec Arguments
```

The `Expression` type describes the soft-typed expressions allowed within the AMSL language. It may also describe an invalid expression, such as `"foo" + 42` or `1 // 0`; the validity of an expression is determined when it is evaluated with the following function:

```
evaluate :: Env -> Expression -> Either InvalidExpression ExpValue
```

An `ExpValue` is a completely evaluated expression. Note the `Env` parameter: it is a simple mapping from identifiers to `ExpValue`'s, and represents the *namespace* in which an expression is evaluated.

The expression namespace contains two kinds of entries: the names and values of enumeration fields, which can be used in the entire module, and the names and values of preceding record fields. In order to help providing the correct `Env` when manipulating record values, a fold and unfold function are provided to respectively consume or produce an `a` according to a `Record a`:

```
-- | Process the fields of a record one-by-one in some monad,
--   while the expression environment is updated accordingly.
foldRecord :: Monad m => Env
               -> (forall t. Env -> (Type t, Arguments)
                    -> (Codec, Arguments) -> t -> m ())
               -> Record a -> a -> m ()
```

```
-- | Construct the value of a record, within some monad, by
--   producing its fields one by one. Updates the expression
--   environment with previously produced field values.
buildRecord :: Monad m => Env
               -> (forall t. Env -> (Type t, Arguments)
                    -> (Codec, Arguments) -> m t)
               -> Record a -> m a
```

4.3.4 Deriving Message Encoders and Decoders

The library includes an important application of the AMSL data types: automatic message encoders and decoders. Given a `Type a` and a corresponding codec, these can parse an `a` from a bitstring, or produce the binary representation of any given `a`. These functions have the following signature:

```
-- | Encoder for a particular type. Should not fail when the
--   value passes the checker, or when it was produced by the
--   decoder.
encode :: TerminatorGenerator -> Env -> (Type a, Arguments)
          -> (Codec, Arguments) -> a -> BitPut ()

-- | Decoder for a particular type. Fails when input is
--   incorrectly formatted or the parsed structures are not
--   permissable according to type arguments in the
--   specification.
decode :: forall a. Env -> (Type a, Arguments)
          -> (Codec, Arguments) -> BitGet a
```

The `BitGet` and `BitPut` monads respectively contain binary parsers and serializers from the `binary-bits` package [37].

The `TerminatorGenerator` provided to the encode function capable of generating values of any given `Type` to be used when generating a final value of a list using the `TerminatedUnionList` codec. This is necessary because this terminator value is not stored in the decoded data type. Generally, one can simply pass in the predefined `arbitraryGenerator` here.

One task of the decoder is to validate whether the parsed value is actually permissible according to the type arguments; e.g. it should not accept the integer value `5` when it's AMSL type is described as `Integer(max=4)`. For that purpose, it uses the following operation:

```
checkValue :: Env -> Type a -> Arguments -> a -> Bool
```

Note that `checkValue` does not receive a `Codec` argument: how a value is coded is not relevant for the decision on whether it is acceptable.

4.3.5 AISL Modules

A module within the AISL language is represented as follows:

```
data Module = Module {
    -- | The name of the module.
    moduleName :: String,

    -- | The messages that are used within this system,
    --   associated with their module environments. Their
    --   names are used as action labels within the LTS's.
    messages   :: [(Message, Env)],

    -- | An LTS associated with each actor described in this
    --   definition.
    actors     :: [(String, LTS)]
}
```

Besides its name, a `Module` object only contains a list of references to AMSL messages, and an `LTS` object for each actor. An `LTS` is a (slightly adjusted) Haskell representation of a labelled decision system as described in Section 3.3; it is derived from the AISL abstract syntax tree using the method described in Section 4.2.4. Its data type is straightforward:

```
data LTS = LTS {
    initState   :: StateID,
    transitions :: Map StateID Transitions
}

data StateID = Normal String | Intermediate Integer | Exit

type Transitions = Map Action [StateID]

data Action = Send MessageLabel | Receive MessageLabel
            | Internal | ReceiveUnexpected

type MessageLabel = String
```

4.3.6 LTS Traversal

In order to explore the effects an actor receiving and sending particular messages should have on its state, the `TraversalT` monad transformer can be used. It describes a traversal of the graph associated with an LTS and has the following primary operations:

```
possibleStates :: Monad m => TraversalT s m (Set StateID)
sent           :: Monad m => MessageLabel -> TraversalT s m ()
received       :: Monad m => MessageLabel -> TraversalT s m ()
```

It can be checked what the currently possible states are by invoking `possibleStates`, while `sent` and `received` are to be called when these events happen. An example of how this can be used:

```
sendAwhenInB = do
    states <- possibleStates
    if mB `elem` states
     then do
        lift $ doSendOperation mA
        sent mA
     else do
        msg <- lift waitForIncomingMessage
        received msg
        sendAwhenInB
```

Here `m` is some monad for handling side-effects that may affect the traversal, and `s` can contain some state that is updated during the traversal (`TraversalT` is an instance of `MonadState`). `possibleStates` provides the potential states the actor might be in at a moment during the traversal; `sent` and `receive` indicate a message has been observed in either direction. Internal actions are assumed to non-deterministically occur at any time, and are handled automatically.

This monad is used during the execution of a test strategy, but it also has other applications: for instance, a tool that allows a user to test a specification by manually indicating which messages are sent or received (a simple version of such a tool is actually implemented as part of the framework), or a passive network listener that explores the LTS while watching the communication of two real systems.

4.3.7 Synchronous and Asynchronous Testing

The current implementation of the `Traversal` monad has a considerable shortcoming: it assumes the modelled system communicates in a synchronous manner, i.e. send and receive actions are performed in the order they appear.

In reality however, the system and its communication partner may send a message to each other roughly at the same time. From the outside, it may be impossible to determine whether the system had performed the send action first and then processed the received message, or vice-versa. How to deal with this additional source of non-determinism has been described in Section 3.3.2, but this has not been implemented in this version of the testing framework.

5 Testing Strategies

In order for a simulator to be able to test a live system, it should consider the following:

- When it is supposed to send a message of a particular type, how to format its contents; i.e. what values to select for its fields.

- Under what circumstance to send what message. Particularly, what it should send in order to test as many states of the system as possible.

This section describes what strategies are used in order to make these decisions, and how those are represented in the framework. The *message generation* strategies are derived from the corresponding AMSL module, and the *LTS exploration* strategies are derived from the AISL specification. These strategies are independent of each other, meaning one of the two could be switched out. Furthermore, they can also be used seperately: for example, when testing a file format, one may not care about the protocol state machine (which simply consists of an application reading the file and responding whether it is correct or not) but would still want to use a standalone AMSL specification in order to generate test inputs.

5.1 Message Generation

A message generator can be asked to provide an instance of a particular kind of message. It has the following type:

```
genMessage :: GenParams -> Env -> Message -> Gen MessageInstance
```

Here, `Env` is the expression environment (see 4.3.3) associated with the AMSL module. The `Gen` type is a monad from the *QuickCheck* project that describes a generator that outputs values of a certain type according to some randomized strategy [25]; because it has a similar purpose, the QuickCheck library offers useful primitives when generating values of AMSL types.

`GenParams` is an object containing configuration options: currently it only contains a *size* parameter, and a related *integer bounds* parameter. Both serve the purpose of limiting the size of outputs: this is important when a protocol allows for messages that may be petabytes long, or even have unlimited length. These parameters do not put a precise limit on the output size, but do influence how certain values are generated, as listed below.

5.1.1 Generating Particular Values

The function `genMessage` is a wrapper around the following:

```
genValue :: GenParams -> Env -> (Type a, Arguments) -> Gen a
```

The `genValue` function provides a generator for any AMSL type, along with its type arguments. Note that the decision to seperate data types and codecs pays off here: the generator builder does not need to consider encoding at all, but can simply provide Haskell values, given that they are allowed according to the arguments. Resulting generators are composable, making it simple to recursively invoke `genValue` in order to build up composite types.

The following heuristics are applied, when generating values of various types (see Section 4.3.2 for a description of these types):

- In case of a type alias, the arguments are updated and `genValue` is simply invoked for the aliased type.
- When an argument fixes the permissible values to only one option, that value is always returned.
- *Boolean* values are simply picked by metaphorically flipping a coin: about half of the results will be `true` and half will be `false`.
- AMSL requires that *Integer* values must always be bounded, so the generator simply picks an integer uniformly within permissible range. However, when this range is greater than what is permissible by the integer bounds parameters, it is restricted.
- For the *List* type, a length is first selected uniformly from the range of permissible lengths (restricted by the size parameter); then, each individual element is generated.
- The *Text* type also first selects a length, and then each code point is uniformly drawn from the given character set (all of Unicode by default). AMSL also allows, however, to specify a regular expression pattern the text field should match; when this argument is present, a string of the desired length will be generated that matches this regular expression. This is achieved by a custom dynamic programming algorithm that returns a list of generators which can each select a string of a particular length; a regular expression such as `A|B`, for example, is handled by recursing on `A` and `B` and then combining these lists in the correct manner. The worst-case complexity of this algorithm is $O(nr)$ where n is the length of the desired string and r is the length of the regular expression.
- The *Binary* type (which represents bitstrings), uses the same mechanism as Text. AMSL even allows the specification of a regular expression which should match the hexadecimal representation of all permitted byte sequences; in this case, a hexadecimal Text value is generated first and then decoded into a bytestring.
- For a record, the generator is folded (see Section 4.3.3) over its fields; i.e. these are generated from top to bottom while updating the environment with the field values established so far.
- For unions, one of the options is selected uniformly; then the function recurses on the type of that option.
- Enumerations are also generated by uniform selection.

5.1.2 Issues

The heuristics for message generation used here have been chosen because they provide a simple and straigthforward method for obtaining varied and correct

messages; however, that does not mean it is perfectly suitable for testing. Section 5.3 describes a method of measuring the effectiveness of a particular strategy and Section 6 uses this measure when trying the generation strategy described here on case studies.

Certain shortcomings of the current testing method can already be identified:

Unbalanced union selection When a union has two members: one with a single constant value and one complex record with many fields, its generator will still produce the constant value about 50% of the time. This is undesirable, because this will lead to the simple value being repeated many times while the other complex value may have many variations that one wishes to test. This problem is exacerbated when many unions are contained in a message, leading to an overrepresentation of the simpler alternatives.

A solution might be to attach a weight to a type that increases when it contains more elements that can be varied (when perceiving fields as tree nodes, this could be the amount of descendants); then, elements with a higher weight should be selected more frequently.

No weighing of 'interesting' values When, for example, having to pick an integer value between 1 and 2^{64}, this generator performs uniform selection. However, certain integers may be of additional interest because implementation errors are expected to be more likely when they occur (particularly the minimal and maximal values may be interesting, due to *off-by-one* errors). The likelyhood that the current generator will actually pick the 'interesting' value 2^{64}, though, is only 2^{-64} and thus negligible.

The test tool may benefit from employing heuristics that pick values from a distribution in which particular values appear with greater probability than others. The effectiveness of these heuristics should be determined by experimentation.

Furthermore, what values are considered interesting may depend on the protocol, so the user could be provided with a method (possibly an extension of AMSL) to indicate specific values they want to test more frequently.

Strategy can not be tweaked by the user As noted above, users are currently not able to specify which values of a field are considered to be more interesting than others; i.e. they can not influence the distribution from which values are drawn.

While it is possible to write an entirely new generator from scratch, there is currently no interface that allows users to design a custom generator which they can attach to a particular field while still using the standard generator by default for all other fields.

A possible solution would be to use a similar approach to T3, a random testing tool for Java [39]. T3 is capable of automatically testing Java classes by generating test suites that perform sequences of method calls with arguments that are varied randomly (in a manner similar to Quickcheck [25]). An extension

of the tool, called T3i, uses this mechanism by default but also allows the user to have a high degree of control over the manner in which test suites are generated: besides allowing users to compose, query and modify test suites, they can *inject* specific generators for particular values [40]. For example, a generator could be specified that, for all integer arguments named *birthyear*, will pick a number between 1900 and 2016.

In the context of AMSL, specific generators could be attached to specific fields or types (or any of them which match a particular predicate) that are to be used instead of the defaults. These generators should be composable and easy to construct, in a manner similar to those in T3i.

Naive record generation Consider the following record:

```
record Foo with
    some_number   is Integer(min=0, max=10000)
    special_field is Optional(subject=Bar,
                              is_empty=some_number != 42)
end
```

When `some_number` has the specific value of 42, the `special_field` will be present, but otherwise it is not. The current generator will first use its `Integer`-generator to pick a value for `some_number`; if this value does not happen to be 42 (a probability of $\frac{9999}{10000}$), then `special_field` can only be empty. This means the `Bar` within the field will rarely, if ever, be tested.

The problem is that the current record generator provides a field value and then fixes it, but does not 'look ahead' at how this field is being used. If the generator would have worked backwards instead of vice-versa, it might instead have decided that it wants the `special_field` to be non-empty, and instead fix `some_number` to be 42.

One method to improve in this area would be for the generator to first examine in which *predicates* a field value is referred to; then, it should try to generate values for this field such that each of these expressions will be `true` or `false` roughly the same number of times.

5.2 LTS Exploration

The `Traversal` monad, described in Section 4.3.6, abstracts over the act of determining an actors' state(s) depending on the actions that have been observed. The *simulator* that performs the actual test is expressed in terms of this monad. The simulator registers which messages it receives and whether errors occur, but it should also determine what messages to send. How it comes to this decision is called the *strategy*, and it has been abstracted over in the implementation:

```
-- | A strategy is defined by how to act at a decision point,
--   and how to update the state. The parameter s represents
--   the state, and r the test report when finished.
```

```
type Strategy s r = ExploreState s -> (Decision r, s)

-- | The state of the exploration process. Based on this a
--   decision needs to be made.
data ExploreState s = ExploreState {
    -- | The complete LTS of the tested system.
    lts :: LTS,

    -- | The possible states the system may be in.
    potentialStates :: Set StateID,

    -- | The latest received message; or Nothing in case there is none.
    receivedMessage :: Maybe MessageLabel,

    -- | Additional user-defined state. It is maintained
    --   after a reset and may be updated at each decision
    --   point.
    userState :: s
}

-- | How to act at a 'decision point' in the testing process.
--   r is a type representing final test results.
data Decision r
    -- | Send a particular message.
    = SendMessage MessageLabel
    -- | Wait until a new message comes in, or a timeout.
    | AwaitMessage
    -- | Finished testing.
    | Finish r
```

A strategy is expressed as a `Strategy` function type, which examines the current state of the LTS exploration (which includes a strategy-specific state variable) and makes a decision on what to do next. See the comments in the code snippet above for a description of the individual data types.

The current implementation assumes systems can be tested *synchronously*, in the manner described in Section 4.3.7. This is not correct for real implementations of certain protocols, which may result in the testing tool reporting false results (or applying its strategy ineffectively) when a tested system is capable of sending and receiving messages at the same time.

5.2.1 The 'Greedy Random Walk' Strategy

Just like the message generator, the implemented LTS traversal strategy uses a basic and simple heuristic. The technique used here I dubbed the *greedy random walk*. It uses the following algorithm in order to determine what message to send:

- Mark any state the system is possibly in as *visited*.

- If there are one or more *unvisited* states that can be traversed to from a possible current state through a send action, randomly and uniformly pick one of these states and send the message needed to move to it.
- Otherwise, if there are one or more visited states that can be moved to with a send action, pick one of these and send the corresponding message.
- When nothing can be send, await message reception.

This step is repeated for an amount of times configured by the user. After that, the strategy terminates.

This strategy basically tries to perform a random walk through the state graph, preferring unvisited above visited states in the hope of covering as much of the graph as possible. A clear shortcoming of this approach is that it does not look past its immediate neighbours: situations in which two or more particular messages need to be send in order to reach an unvisited node are ignored.

Do note that when the system may be in more than one state at the same time, all possibilities are marked as *visited* even though it is uncertain this did indeed happen. Another approaches would be to only mark nodes that have certainly be reached or to introduce an additional class of *possibly visited* nodes that are treated differently.

Thanks to the `Strategy` abstraction layer it is, however, relatively simple to adjust this strategy or to replace it with an alternative one that uses a more intelligent approach.

5.3 Measuring Test Coverage

In order to be able to compare the effectiveness of different testing strategies, a method is needed to measure the degree to which they *cover* the specification: i.e. what elements of the protocol, that the specification writer has described, are being subjected to the test. Of course coverage by itself does not conclusively say how good a strategy is at its primary goal, namely finding implementation bugs; it can, however, be useful for comparing strategies and as a measure of how thorough a test was that did not find flaws.

Methods have been defined to measure testing coverage of respectively AMSL and AISL modules. This measurement provides information about how effective a strategy is at testing a protocol with one particular specification; however, those results can not be extended to all protocols. It may also very well be possible that strategy A is better than strategy B when testing a particular specification, while strategy B is superior when testing another.

5.3.1 AMSL Coverage

AMSL specification coverage is treated similarly to the concept of *code coverage*: the goal is to measure how many *statements* made by the specification writer are *reached* during execution. In this context, I define a statement to be a field within a record, union, or enumeration and I consider it to have been reached

once this field is present somewhere within a message. Additionally, *Optional* fields are only considered to be fully covered when they have been encountered while empty and when containing a value.

The framework defines a `Coverage` data type that contains a binary tree associated with a single message specification, along with two functions to build it:

```
data Coverage
    = UncoveredLeaf
    | CoveredLeaf
    | Node Coverage Coverage
    | Label TypeName Coverage

-- | Determines how much a specification of a single message
--   covers.
messageCoverage :: MessageInstance -> Coverage

-- | Join the coverage of multiple instances of the same
--   message or identically labelled structures.
combineCoverage :: Coverage -> Coverage -> Coverage
```

For each instance of the same message, the structure (i.e. placement of `Node`s and `Label`s) of coverage trees will be identical; the leaves differ depending on which fields were present in the instance and which were not. The `combineCoverage` function merges structurally identical trees: a covered and an uncovered leaf are combined into a `CoveredLeaf`.

Composite types (records, unions and enumerations) are represented as nodes. When *painting* (marking as covered) a record, all its fields are recursively painted; when painting a union or enumeration, only the option actually present is painted.

`Label`s attach the type name of a user-defined composite type to a subtree. Two subtrees with the same label as their root must be structurally identical, because they express the coverage of the same type. These are necessary because when instances of the same type appear at different places they should not be counted separately: a field is already considered covered when it appears in any location. Therefore subtrees with the same label are also automatically merged with each other.

The following functions can be used when willing to express the coverage of an entire AMSL module, rather than that of a single message:

```
-- | 0% module coverage that can be extended with
--   moduleCoverageAdd. The argument contains the messages in
--   a (AMSL or AISL) module.
moduleCoverageBase :: [Message] -> ModuleCoverage

-- | Add the coverage of a particular message to the total
```

```
--   module coverage.
moduleCoverageAdd :: MessageInstance -> ModuleCoverage
                                     -> ModuleCoverage

-- | Cover fraction for all messages.
moduleCoverAmount :: ModuleCoverage -> Rational
```

The `ModuleCoverage` data type maintains the `Coverage` for each message and for each other type (identified by a `Label` within a coverage tree). With `moduleCoverageAdd` these coverage trees are appropriately painted based on a single message instance.

The `moduleCoverAmount` function attaches a score between 0 and 1 to a coverage result: namely the fraction of leaves that are covered, among the total number of leaves within a coverage tree.

Message coverage could be measured under different circumstances: for example, the message generator could simply be tasked to generate a certain number of instances for each message, and a `ModuleCoverage` could be derived from that. For the case studies in Section 6, the AMSL coverage is computed over the messages that are send to the tested system, but also over those that are received from it. That means that the coverage does not just say something about the generator, it also gives information about what parts of the protocol the other system can be forced to utilise.

5.3.2 AISL Coverage

For an AISL actor, the coverage measure is relatively simple: namely the number of edges within the derived LTS that have been traversed. Each edge is represented by a $(state, action, state)$ triple, and the coverage is defined as the fraction of them that are crossed during testing.

Since every statement that is added to an AISL specification causes new edges to be added to the derived LTS (see Section 4.2.4), a strong edge coverage implies strong "code coverage" of the AISL module.

A problem arises in the case of non-determinism: when it is unclear whether the system is in state A or state B, and both states have an edge to state C that is triggered when receiving message m, then it is unclear which edge is traversed when m is observed: it could be either (A, m, C) or (B, m, C). Here, I decided to pessimistically consider neither edge as covered (it is impossible to determine which is the correct one anyway); an alternative approach would be to count both, or to divide fractional coverage scores among them.

Coverage is automatically maintained within the `Traversal` monad described in Section 4.3.6:

```
type Transition = (StateID, Action, StateID)
newtype Coverage = Coverage { unCoverage :: Set Transition }

-- | Get the total coverage. In case of nondeterminism only
```

```
--   the transitions of which it is certain that they have
--   been touched will be included.
getCoverage :: Monad m => TraversalT s m Coverage
```

The `Coverage` data type simply contains a set of transitions (edges). The fraction of these, out of the total, provide a coverage score.

5.4 Alternatives to Random Testing

When testing a protocol specification, the number of possible test sequences and message variations increases exponentially with any addition to the specification, therefore it is generally infeasible to use an *exhaustive testing* approach, in which every possibility is tried. Instead, both testing strategies given in this section employ a *random testing* approach: when faced with multiple options, the next step is determined randomly and through repetition it is hoped that a large variation of cases is examined.

However, alternative methods could also be employed when designing a message generator or LTS traversal strategy, such as *search-based* or *combinatorial* testing:

Search-based testing By using a coverage measurement such as the one described in Section 5.3 and applying it to the result of executing a strategy for several test protocols, one can attach a number to the quality of a testing strategy and compare the effectiveness of different strategies. Coverage can also be used as a *fitness value* within the context of search algorithms, when the subject to be optimized is the strategy itself: by representing the test strategy as a set of parameters that can be (selectively) varied, one could run a search algorithm to try and maximize the coverage/fitness score [43].

An example of this approach can be found in the EvoSuite tool [44], which generates test cases for programs written in the Java programming language. Instead of generating arbitrary tests at random, it starts with a population of random strategies and applies and uses a genetic algorithm to evolve them until one has a high enough fitness score (which is defined in terms of code coverage). [45] demonstrates this tool can compete well with others that use a random testing approach.

Combinatorial testing When using exhaustive testing, one tries every possible configuration of all parameters, of which usually too many exist for this to be practical. *Combinatorial testing* is based on the observation that usually faults occur as the result of the interaction of a small number of these parameters; based on this, a combinatorial testing strategy looks at all n-way combinations (where n is a small number, in practice never higher than 6; when $n = 2$, the approach is called *pairwise testing*) of parameters and tests all values of every combination, while leaving the other parameters constant [46].

For example, when given 10 parameters which each have 20 values, exhaustive testing would require all $20^{10} \approx 10^{13}$ configurations to be tried; however,

when using 3-way combinatorial testing, only $\binom{10}{3} * 20^3 = 960,000$ possible configurations exist. Furthermore, a single test case with 10 may be particular parameter values will able to cover multiple combinations of three parameters at once.

In order to determine the minimal amount of test cases that are needed in order to cover all 3-way combinations, one can look at a minimal *covering array* of configurations: a list of combinations to test that together cover all configurations [51]. For the example, where one wishes to cover all 3-way tuples of 10 parameters that each have 20 different values, a covering array exists that only has size 8930; which means only that many test cases will have to be executed [52].

This approach might be suitable for AMSL message generation, where the parameters to test are the values of fields (and sub-fields) within a message.

6 Case Studies

In order to test the languages and the testing framework so far, two protocols have been picked as case studies: first, (a subset of the) protocol messages have been specified in the AMSL language; then, an AISL model was constructed for a protocol role (in both cases the *server* role). The testing framework was used to automatically execute tests by using the strategies described in Section 5 (causing the tester to assume the role of *client*), while measuring test coverage.

6.1 WebSocket

The WebSocket protocol provides a message-based bidirectional communication channel, layered over TCP. It is aimed at browser-based applications that wish to efficiently do two-way communication with a server without opening multiple HTTP connections. In order to work around firewalls blocking non-HTTP traffic, the protocol uses the HTTP ports (80 and 443) and starts with a handshake that is identical to an HTTP *protocol upgrade* request and response pair. The protocol is specified in RFC 6455 [47], and this RFC was followed in order to design the AMSL and AISL specifications discussed here.

WebSocket is an interesting case study because its structure is relatively simple and easy to test (one can easily set up a websocket server; furthermore, the content of the messages can be arbitrary and do not need to have particular semantics, as long as they are framed correctly); yet, it does have some interesting properies: such as the transition from the HTTP-like handshake to the binary framing messaging phase, and the manner in which payload length is determined.

6.1.1 AMSL Specification

Frames I initially identified three types of messages within the Websocket protocol: a *client handshake*, a *server handshake*, and a *frame* containing (part

of) a payload one party wishes to send to another. Then, when trying to write the APSL specification, I discovered it to be neccessary to distinguish different kinds of frames: this is because of two properties within a frame, the *opcode* and the *fin-bit*. When fin is 1 it indicates that the frame contains the last part of a WebSocket message (I will use the term *WebSocket message* to refer to a sequence of related frames within the WebSocket protocol, which should not be confused with an AMSL message), while a fin of 0 indicates a partial WebSocket message that is to be completed by following frames.

A WebSocket message consists of zero or more frames with $fin = 0$, followed by one final frame with $fin = 1$. In such a sequence of frames (called a *continuation* by the standard), the *opcode* property of the first one indicates the type of the WebSocket message, while all the following frames are required to have an opcode of 0. Because the position of a frame within a continuation can depend on the frames received previously, I decided to define four messages for different kinds of frames:

- A `MessageStartFrame`, which has a nonzero opcode and a fin-bit set to 0.
- A `ContinuationFrame`, with the opcode and fin-bit both being zero.
- A `MessageEndFrame`, with a zero opcode and a fin-bit of 1.
- A `MessageFrame`, which has a nonzero opcode and a fin-bit set to 1; it represents WebSocket messages which are encapsulated in a single frame in their entirety.

To prevent replication, all these messages simply contain one `Frame` record; this record has boolean parameters called `first` and `final`, and each message type listed above uses a different configuration. The record is defined as follows:

```
record Frame(first, final, must_mask) with
 fin           is Bool(value=final) as Bit
 _reserved     is Binary(value=b'000')

 opcode        is Optional(subject=Opcode, is_empty=!first)
                   as OptionalCodec(subject_codec=BE(length=4),
                                    null_string=b'0000')
 mask is Bool(value=true if must_mask else null)
          as Bit
 payload_len   is Integer(min=0, max=127)
                   as BE(length=7)
 payload_len16 is Optional(is_empty=payload_len != 126,
                           subject=Integer(min=126))
                   as OptionalCodec(subject_codec=BE(length=16))
 payload_len64 is Optional(is_empty=payload_len != 127,
                           subject=Integer(min=2^16))
```

```
                    as OptionalCodec(subject_codec=BE(length=64))
  mask_key      is Optional(is_empty=!mask,
                            subject=Binary(length=32))
                    as OptionalCodec(subject_codec=FixedLengthBinary)

  payload_data  is Payload(
                    tag=opcode if opcode != null else bin_frame,
                    length=8 *
                     (payload_len   if payload_len <= 125 else
                      payload_len16 if payload_len == 126 else
                      payload_len64))
end
```

Some noteworthy aspects:

- The *opcode* field is an indicator for the type of frame: a `text_frame` or `bin_frame` has a regular payload that is marked respectively as containing text or arbitrary bytes. A `ping` frame requires the server to respond with a `pong` frame containing the same data. Another approach to specifying the protocol would be to define different message types for various opcodes, instead of distinguishing between frame positions within a continuation. I chose to focus on representing continuations, although it would be possible to also define different messages for the different types of start frames by extending the specification even more.
- The length of the payload (in bytes) is determined by a 7-bit field. However, when it has the special value of 126 it is followed by another 16-bit field containing the actual length; when it has the value of 127, the additional length field is 64-bit. This structure is represented by using `Optional` fields and expressions depending on the value of `payload_len`.
- When the `mask` bit is set, the value of `mask_key` should be XOR'ed with every four bytes within the payload. The purpose of this is to prevent certain protocol confusion and cache poisoning attacks on incorrectly implemented HTTP servers [47]. While AMSL does not currently support a method of expressing how the payload is transformed it is safe to ignore it during these tests, since the actual value of the payload is unimportant.
- When the `must_mask` parameter is `true`, masking must be used. Otherwise, it may or may not be activated.

The reason the `must_mask` parameter is included is that the specification requires that WebSocket clients must always mask their messages, however servers may freely choose whether or not to mask and could even choose to mix masked and unmasked messages within the same session. In order to enforce clients to mask while allowing servers to choose whether to do this or not, I introduced four additional frame messages: they correspond to the four defined so far, with the difference that they set `must_mask` to `true`.

The handshake The client handshake message, which is the first thing being send when establishing a WebSocket connection, looks like an HTTP request with a particular set of headers. I defined it as follows:

```
record HttpHeader(key, val) with
    name  is Text(pattern=key, exclude_pattern=/:/)
             as TerminatedText(encoding='ascii',
                               terminator=':')
    value is Text(pattern=val, exclude_pattern=/\r\n/)
             as TerminatedText(encoding='ascii',
                               terminator="\r\n")
end

message ClientOpenHandshake with
    _method  is Text(value='GET ')
    resource is UriPath(exclude_pattern=/ /)
                as TerminatedText(encoding='utf-8',
                                  terminator=' ')
    protocol is Text(value='HTTP/1.1')
                as TerminatedText(encoding='utf-8',
                                  terminator="\r\n")

    host         is HttpHeader(key=/host/,
                               val=/[a-zA-Z0-9\.]+/)
    upgrade      is HttpHeader(key=/upgrade/,
                               val=/websocket/)
    connection   is HttpHeader(key=/connection/,
                               val=/upgrade/)
    key          is HttpHeader(key=/sec-websocket-key/,
                               val=/[A-Za-z0-9\+\/]{22}==/)
    version      is HttpHeader(key=/sec-websocket-version/,
                               val=/13/)

    _end         is Text(value="\r\n")
end
```

Even though AMSL is not intended to be used for text-based protocols, the use of text fields and regular expressions still enabled the construction of this particular kind of HTTP request. Unfortunately, when testing this, I found out I missed an important aspect of the standard: namely, that it is allowed to add additional arbitrary HTTP headers, and that they could be permuted in any way.

Allowing arbitrary headers would not be difficult to represent, but I could not find a method to practically represent this permutation, and have not yet succeeded in adding language features to AMSL that would circumvent this shortcoming. Another method of representing HTTP headers, would be to

simply include a single text field covering all of them; however, that would allow a wide range of invalid header sets that would immediately be rejected by a server when generated.

Because of this, I went with a compromise solution: `ClientOpenHandshake` would be represented as demonstrated above, and `ServerOpenHandshake` was given the following definition:

```
message ServerOpenHandshake with
    status_line is Text(pattern=/HTTP\/1\.1 101 [ -~]*/,
                        exclude_pattern=/\r\n/)
                    as TerminatedText(encoding='ascii',
                                      terminator="\r\n")
    headers is Text(exclude_pattern=/\r\n\r\n/)
                as TerminatedText(encoding='ascii',
                                  terminator="\r\n\r\n")
end
```

Now the definition of `ClientOpenHandshake` is too strict, while that of `ServerOpenHandshake` is too liberal. This will still allow us to perform sound tests of WebSocket servers, though, as generated client handshakes will be valid and correct server responses are accepted. Unfortunately, testing clients will no longer work: clients that do not use this particular order of headers will result in a false positive and they would probably never accept a randomly generated server handshake.

6.1.2 AISL Specification

The internal state machine of a WebSocket server is quite simple: initially it is in a *connecting* state, waiting to receive the client handshake. When it accepts it, it will send its own part of the handshake and move to the *open* state, in which it can start sending and receiving messages. These two states are specified as follows:

```
actor WebsocketServer with
    init state Connecting where
        on ClientOpenHandshake do
            send ServerOpenHandshake
        next Open
    end

    state Open where
        on MaskedMessageFrame do
        continue

        on MaskedMessageStartFrame do
        next ReceivingContinuation
```

```
        anytime do
            send MessageFrame
        continue
        or do
            send MessageStartFrame
        next SendingContinuation
        or do
            send MessageFrame
        quit
    end

    .....
```

Handling frames that do not use continuations is straightforward, but after sending or receiving a `StartFrame`, it should be followed up by zero or more `ContinuationFrame`s, ending with a `MessageEndFrame`. I encoded this by introducing three additional states that behave in the same manner as `Open`:

- In the `SendingContinuation` state, the server is capable of sending either a `ContinuationFrame` (and staying in the same state), or a `MessageEndFrame` (followed by a transition back to `Open`).
- In the state `ReceivingContinuation`, the server expects continuation frames or an end frame from the client.
- Finally, when the server receives a continuation in the `SendingContinuation` state, or sent one during the `ReceivingContinuation` state, it will transition to `SendingReceivingContinuation`, in which it will both send and expect continuations.

6.1.3 Test Setup

The server being tested was a simple *echo server* (i.e. a server that replies to each message by repeating its content), using the Autobahn WebSocket library [48]. The strategies used were those described in Section 5. The *greedy random walk* strategy was configured to take at most 500 steps.

This server was configured to accept connections on the local machine on port 9000. Unfortunately, this caused a problem: the server would never accept a randomly generated `Host` header within the client handshake, because it strictly enforces that the provided hostname matches the one assigned to itself.

While the hostname of the server is configurable, it could not be instructed to ignore the one provided by the client. Therefore, the AMSL specification was altered and the host header was simply fixed to `localhost:9000`. While this solution is fine in the context of this test, it prevents the AMSL specification of the protocol from being generic and applicable in other situations.

This points to a shortcoming in the framework: ideally, it should be easy for users to configure tests in such a way that it would be possible to fix certain fields

to specific values. Currently, this is only possible by either altering the AMSL specification or by writing a Haskell function that fixes this field and applying it to all outgoing messages (using the framework to intercept and reconstruct them).

6.1.4 Problem: Aynchronous Communication

At the moment of testing with this specification, I was not yet aware of the difficulties of modelling asynchronous protocols (i.e. those in which the parties may send messages at the same time) described in Section 3.3.2. I implicitly and incorrectly assumed that the current test framework would be able to correctly model the traversal of a `WebSocketServer`'s LTS.

However, as explained in Section 4.3.7, the system is currently not sound when applied to asynchronous protocols, and because a WebSocket server is capable of sending and receiving messages at roughly the same time (as can be seen in its specification) this caused a problem: while no false positives were yielded, the tester would never wait for send actions (i.e. the subject sending to the tester) to occur when it could also trigger a receive action (i.e. the subject receiving a message from the tester); because a strategy was constantly decided before the tested server had a chance to respond, the system would end up solely transmitting messages and triggering input actions, meaning many parts of the state machine were never covered.

This problem was solved by introducing a short timeout when a send action could possibly occur: if a message would come in within that time, it would be handled; otherwise, the tester would move on to trigger a receive action. Because the test subject is a simple echo server capable of responding withing the timeout interval, asynchronous communication was avoided and the tester behaved correctly. However, this technique is not generally applicable, not even to the WebSocket protocol. Furthermore, it caused the execution time of the strategy to be much longer, because of the many timeouts.

6.1.5 Results

The tests were run succesfully on the echo server, given the caveats that have been noted above. The coverage score was determined using the metrics described in Section 5.3. The precise coverage reports can be found in the file `test-cases/websocket-results.txt`. The results are as follows:

AMSL Message Coverage The message coverage score was 65%. Upon closer examination, the incompleteness of the message coverage was caused entirely by two factors:

- None of the messages `MessageStartFrame`, `ContinuationFrame` or `MessageEndFrame` were ever encountered. Their masked variants were, however. The reason for this is that the echo server being tested simply never utilised multi-frame WebSocket messages, only the client did.

- All fields in all other messages were covered, with the one exception of the optional field `payload_len64`, which was always empty.

The fact that the server continuation frames were missed is caused by this particular server implementation, and (assuming that it could not have been forced to use the continuation feature) does not say anything about the testing strategy, since any strategy would have had this same result. Note that even when certain parts of the specification are never covered, such an experiment can still be useful: it still allows strategies to be compared on how well they cover other aspects of the protocol.

The fact that the `payload_len64` field was missed does point to a shortcoming of the message generation strategy, though: because the value of the `payload_len` field is selected uniformly between 0 and 127, it is very well possible that the significant value of 127 never happened to be selected, resulting in the 64-bit field to never be included. This problem is discussed in Section 5.1.2.

AISL Transition Coverage The transition coverage score was 28.57%. While this score seems low, it is entirely caused by the server never sending any continuation frame and thus never entering the `SendingContinuation` or `SendingReceivingContinuation` states. Transitions from, to, and within these states take up the majority of the specification; when they are disregared, the coverage result becomes 100%: all other states are covered.

This is not very surprising: because the remaining state machine is very simple, running a randomized traversal long enough would quickly have encountered all transitions even by accident. A test subject that would employ the continuation feature of WebSockets would probably have yielded more interesting results, when it came to testing this particular APSL specification of the protocol.

It can be concluded that this case study does not give very useful information about the *greedy random walk* traversal strategy. However, its message coverage results are interesting, and the process of designing the APSL specification of this protocol also revealed some interesting points.

6.2 IMAP

The IMAP (Internet Message Access Protocol) is used by e-mail clients to access and manage messages stored remotely on an IMAP server. Among other things, messages can be searched, examined, deleted and organized within different *mailboxes* (which themselves can be created, deleted or renamed). The version of the protocol used for this case study is *IMAP version 4rev1*, as defined in RFC 3501 [49].

In my opinion, IMAP is much less clearly specified than the WebSocket protocol: certain details (such as the allowed characters within and maximal length of *command tags* and identifiers) are not specified, requiring assumptions (based on the apparent behaviour of existing implementations, which are not always consistent with each other) to be made during the design of the AMSL

and APSL specifications. I would like to argue this shows another benefit of protocol specification languages such as APSL: designers are forced to think about ambiguities within a natural language specification, revealing flaws within that specification that require clarification.

Because IMAP is relatively complex, and has many features, only a subset has been specified and tested for this experiment. The features that are included are:

- The three different *server greetings*, that respectively indicate that the user is required to log in, that they are already authenticated and can proceed directly, or that they have no access and the connection will be closed immediately.
- Logging in with a username and password.
- Examining, creating, deleting and renaming mailboxes.
- Selecting a mailbox, after which messages within can be fetched, altered or copied. The possibility to fetch specific parts of an e-mail is not included, only three options: retrieving the entire message, obtaining only the *envelope* (metadata) or just getting the message size and *flags* (properties such as an e-mail having been read or being a draft).
- Closing a selected mailbox, allowing the client to continue opening another.

A notable omission is the option to add a fresh new message to the mailbox; the only means through which it is possible to add new e-mails with these functions is to copy an existing one.

6.2.1 AMSL Specification

Similarly to SMTP and POP3, IMAP messages consist of human-readable ASCII and can actually easily be typed in manually. These do not require complex parsing however, and proved not to be that difficult to express in AMSL.

Within IMAP, messages send from the client to the server are called *commands* and messages in the other direction are referred to as *responses*.

Client commands Each command starts with a *tag*, an arbitrary string of alphanumeric characters chosen by the client, followed by a command name and then arguments of which the type depends on the command. These parts are separated by spaces, and the command always ends with a line terminator. I expressed the initial two fields within the following record:

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

```
    end

    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=' ')
    codec LineTerminated is TerminatedText(encoding='ascii',
                                           terminator="\r\n")
```

Each command that takes arguments is then represented as an AMSL message that starts with a `CommandStart` (the record parameter containing the name) and is followed by a field for each argument. The final argument field either uses the `LineTerminated` codec, or an additional field matching `"\r\n"` is added at the end. For example:

```
message RenameCommand with
    _start   is CommandStart(command='RENAME')
    mailbox  is MailboxId as SpaceTerminated
    new_name is MailboxId as LineTerminated
end
```

The `CommandStart` record is not suitable for commands that do not take arguments, since these should not be ended with a trailing space. For these, a `SimpleCommand` record was introduced that is identical to `CommandStart`, except that the `name` field uses a `LineTerminated` codec.

Server responses After sending a command, the server may first react with zero or more *untagged responses*, which are prefixed with a `*` character and provide relevant information depending on the command. Finally, it will send a *status response* indicating whether the commanded operation succeeded; this response starts with the same command tag as which the client used.

It is required that the server uses the same command tag as the client did; unfortunately, that can currently not be expressed within APSL. When testing servers (as in the case of this experiment), this is not problematic, since the tester could simply accept any tag and will not yield a false positive. However, testing a client that strictly enforces that the server sends the correct tag (or in which the use of an incorrect tag triggers a bug) may prove to be difficult.

The tagged responses are called *status responses*; each includes an identifier followed by an arbitrary server message that may be useful during an interactive session or when reading logs. They are expressed in AMSL with an enumeration, and by wrapping a generic record:

```
    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'
    preauth as 'PREAUTH'
    bye     as 'BYE'
end

message OkResponse with
    resp is StatusResponse(response=ok)
end
message NoResponse with
    resp is StatusResponse(response=no)
end
message BadResponse with
    resp is StatusResponse(response=bad)
end
```

When a command has been completely processed, the server will send a status response of `OK` (indicating success), `NO` (indicating the command could not executed for some reason, such as a mailbox not existing), or `BAD` (meaning the command received from the client was not understood or incorrectly formatted).

The response ID's of `PREAUTH` (a *server greeting* indicating the client is already authenticated) and `BYE` (sent when logging out) are used within certain untagged responses, which are expressed in the record `UntaggedStatusResponse` that is identical to `StatusResponse`, except that it expects an asterisk instead of a tag. Untagged `OK` and `NO` responses can also be send to provide more intermediate information about an operation.

Other untagged responses always start with a message number (since they always relate to a specific e-mail), and are followed by an identifier indicating their type. They are expressed by wrapping messages around the following record:

```
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
```

Mailbox names Mailboxes can have arbitrary names, but when they are randomly generated it is unlikely that a mailbox happens to exist with the same name, meaning all operations on mailboxes would probably fail. While it is useful to test these failures one would of course also want to continue testing valid mailboxes. Luckily, one can be certain that a mailbox named `INBOX` exists, since IMAP requires that. If it were possible that `INBOX` were an interesting value that should be tested just as frequently as random names, this problem could be resolved. As noted in Section 5.1.2, this feature is not directly supported; however, the same effect can be achieved by letting mailbox identifiers be generated from a particular regular expression:

```
type MailboxId  is Identifier(pattern=/INBOX|[0-9a-zA-Z\-]+/)
```

Due to the manner in which values are generated from regular expressions, the string `'INBOX'` will be selected half of the time. Of course, this is an exploitation of an implementation quirk, and is not guaranteed to work when using a different generator or implementation. A better solution would be to provide a method of configuring the generator in such a way the user could tell it that `'INBOX'` is an interesting value for this type and should be tried frequently.

Message identifiers E-mails within a mailbox are identified by sequential integers from 1 onwards. This can be expressed in AMSL (with an assumed upper bound, since the IMAP specification does not provide it) but, due to the manner in which integers are selected, would result in numbers being chosen that are larger than the amount of messages in the mailbox; which means all operations on messages would fail.

Once again, this problem could be solved by implementing solutions for the issues described in Section 5.1.2. For this experiment, I avoided the problem by setting the maximal message number to 100, somewhat limiting the scope of what will be tested.

6.2.2 AISL Specification

IMAP has an interesting state machine that is more complex than that of the WebSocket protocol. However, the protocol is completely synchronous, avoiding the issues described in Section 4.3.7.

The IMAP specification actually describes the state machine pretty clearly. It works as follows:

- Upon establishing a connection, the server will start by sending a *server greeting*, which has the form of an untagged response. This can either be an `OK` response (after which it moves to the *not authenticated* state), a `PREAUTH` response (triggering an immediate transition to the *authenticated* state), or a `BYE` response (after which it will close the connection).
- In the *not authenticated* state, the server can receive a request to use opportunistic encryption (not considered in this case study) or the client can attempt to authenticate themselves using different methods. The only method supported here is a simple `LOGIN` command with a username and password; when it succeeds, the server moves to the *authenticated* state.
- Within the *authenticated* state, the server can receive commands that relate to the management of mailboxes. After receiving a successful `SELECT` command on a particular mailbox, a transition is made to the *selected* state.
- From the *selected* state, various operations on the mailbox are supported. When a `CLOSE` command is received, the server will return to the *authenticated* state.

The AISL specification is a straightforward translation of this state machine, with three additions: the states `Examining` (after receiving an `EXAMINE` command), `Selecting` (in between the authenticated and selected states), and `Processing` (upon receiving a `FETCH`, `COPY` or `STORE` command). These states represent the process in which a server keeps sending an arbitrary amount of untagged responses before ending with a status response and then moving back to one of the states listed above. The `Selecting` state, for example, is defined as follows:

```
state Selecting where
    anytime do
        send FlagsResponse
    continue
    or do
        send ExamineResponse
    continue
    or do
        send UntaggedOk
    continue
    or do
        send OkResponse
    next Selected
end
```

In IMAP, it is possible for the client to send a `LOGOUT` command from any state, triggering a `BYE` response from the server, followed by it closing the TCP

connection. While this command is simple to represent, I did not include it in this subset: the reason is that the testing framework is currently not capable of restarting the testing process when an exit state is reached, meaning any test would too quickly result in an exit, preventing it to continue.

6.2.3 Test Setup

The IMAP server tested for this case study was version 4.10.0 of the *Courier IMAP Server* [50]. It had been setup with a single user named `imap-test`, who has a password identical to its username. For other settings, its default configuration was used.

Before every test, the mailbox used for this user was cleared and then filled up with 80 messages (note that message identifiers used within this case study's AMSL specification have a range between 1 and 100, meaning most but not all operations on them would succeed). These messages initially have no special flags set and each contain just the 18-byte string `"This is a message."` within their body. This process is automated by a script (or rather, a text file containing a sequence of IMAP commands) called `populate-mailbox.imap`.

Once again, the test was configured to take at most 500 steps.

6.2.4 Deleting the Inbox: A Bug Within Courier?

The IMAP specification forbids the special mailbox called INBOX to be deleted and any attempt to do so should result in the server responding with `NO`. This is exactly what the Courier Mail Server did when the tester tried to do this illegal delete operation. However, it appeared that afterwards, any attempt to select the inbox would fail for an unclear reason. Manually recreating the mailbox would solve the problem, and any operation on INBOX would work as expected until a delete command on it was tried.

It appears that a bug in the Courier IMAP server might have caused this issue. That would mean that the testing framework has successfully led to the discovery of a bug. Do note, however, that this potential bug was not flagged by it but rather discovered by me examining a coverage report and running experiments.

Unfortunately, not enough time was available to investigate whether this issue was indeed caused by a bug in Courier, and what caused it. In the following test (of which the results are listed here), I excluded the option to receive `DELETE` commands from the server specification, such that the tester would no longer trigger this situation.

6.2.5 Results

The test did not report any other errors. The coverage report of this experiment can be found in `test-cases/imap-results.txt`. I made the following observations based on it:

AMSL Message Coverage The total message coverage score was 67.6%; the following aspects were not covered:

- The message `BadResponse` was never covered. The reason for this is that such a response indicates a client protocol error; when the client acts correctly, this should never occur. Since no errors were reported, it is obvious that this message could never have been sent.
- The message `Bye` was also not covered‘. The only situation in which this message could be send (since the client's logout command is not used within this experiment), is when the connection opens and the server rejects it for some reason. Since the mail server had no reason to do so, this never happened.
- Likewise, `PreAuthGreeting` was never sent because no situation of pre-authentication was present. This message not occurring is as expected.
- Within the `StatusResponseId` enumeration, the members `bad`, `preauth` and `bye` remained uncovered, because their corresponding messages were never encountered.
- The message `StoreCommand` never occurred as well. The reason for this is discussed in the next paragraph.
- A `FETCH` command is accompanied by an indication of what parts of a message should be retrieved. This argument is represented here as the enum `MessageDataItems`, with three member called `all`, `fast` and `full`. Of these, `all` and `fast` were never covered. By inspecting the message trace, it seems that the fetch command was simply not executed frequently enough (i.e. twice) and that in both cases the generator happened to produce a `full` value. The reason why it was infrequently repeated is probably because the traversal strategy gives a lower priority to transitions it has already visited, and because many other paths are possible: the random walk happened to no longer cross the `FETCH` transition after the second attempt.

AISL Transition Coverage The transition coverage score was 72.58%, for the following reasons:

- The lack of server greetings other than `OK` meant the pre-authentication and immediate logout paths were not followed.
- The rename command never succeeded. This was due to it being illegal to rename the inbox, or to rename another mailbox to INBOX. Renames involving two random names also never succeeded because (as could be expected) no mailbox with the first given name ever existed.

- It was never decided to send a `StoreCommand`. When rerunning the test, or when increasing the step count, this transition did occur. One could say that this simply means the test did not get to run long enough (note that the currently configured step count is 500), especially since many attempts at commands failed due to a nonexistant mailbox or message number. However, a shortcoming of the *greedy random walk* is also revealed by the message trace: after entering the `Selected` state for the first time, the tester tried a `FetchCommand`, a `CopyCommand` and then a `CloseCommand`. Afterwards, the `Selected` state was never entered again because the tester kept trying commands other than `SELECT`. If the strategy was more intelligent and capable of 'looking ahead' more than one step, it would have been detected that a `STORE` command had not been tried yet, but that a `SELECT` would be needed first before being capable of that.

7 Future Work

Many limitations and shortcoming have been identified within the APSL languages and the testing framework so far. Additionally, there are a lot of possible directions this framework and specification language could be taken in. A number of possible subjects for future work include:

- The AMSL language and the message generator could be extended in ways that solve the issues described in Section 5.1.2.
- While the AISL language offers a useful representation of an LTS model, this model may not be sufficient in order to properly test various protocols. This is mostly because it is currently not possible to reuse detailed information from received messages (i.e. field values) in messages that are send in response. In order to model the TCP protocol, for example, it is required to put an increasing sequence number in each segment, which can currently not be expressed. This was also a (minor) problem during the IMAP case study, since it is not possible to let the server repeat a tag found in a preceding client command. One should consider, however, that such extensions to the language may complicate automated testing considerably.
- As noted in Section 4.3.7, the tester is currently not capable of correctly testing asynchronous protocols. It could be extended to use the techniques described in Section 3.3.2.
- Operations based on AMSL specifications can be expressed by writing a Haskell function that pattern matches on GADT's representing AMSL types. It would be interesting to research whether these could be written in a simpler or more concise manner by using the UU Attribute Grammar Compiler [41]. Considering AMSL types are effectively trees, attribute grammars may offer an intuitive representation for folds over such trees.

- Section 4.1.7 describes a mechanism for AMSL type and codec extensions, which could considerably increase its expressive power. This feature has not yet been implemented, and it would need to be decided what would be a convenient way for a user to supply *plug-in code* that the tester can use in order to work with such an extension.

- The currently implemented testing strategies for messages and LTS traversal are still rather simple, and more strategies can be developed, and then compared using the coverage metric. It may be particularly interesting to examine other techniques than random testing, such as those described in Section 5.4.

- The tester currently works by transmitting valid messages that are supposed to be acceptable by the tested system at that moment. However, many bugs and security issues are triggered by such a system incorrectly accepting and processing invalid messages. It would be useful to have the tester occasionally send an invalid message and examine if the system correctly handles this error; i.e. *negative tests* should be performed.

- The current version of the APSL compiler does not yet correctly validate specifications it parses. It may accept, for example, invalid type arguments, which may result in a crash while testing. Furthermore, when an error is found, the system does not give very useful error messages that can aid in locating the problem. Since the whole purpose of the system is to locate bugs, this is very important in order to make the system usable in practice.

- Here, the focus of the APSL language has been conformance testing. However, there are many other potential applications of this protocol specification language, such as those listed in Section 1. A particularly useful application, for which the language is already very suitable, would be automatic code generation: based on an APSL module, a "skeleton" implementation could be provided that implements message parsing, validation and serialization, along with the protocol state machine. The user would then only have to fill in the gaps with the detailed logic behind the protocol.

8 Conclusion

This thesis discussed how implementers of a system that uses a communication protocol could benefit from a formal protocol specification language, especially when conformance tests can be derived automatically from a specification.

Protocol specification languages that are currently available have been examined, along with the degree to which they would be useful for this purpose. Techniques for automated testing, based on some specification, have also been discussed.

The APSL language has been introduced, which tries to solve this problem. It is actually subdivided into two different languages: AMSL and AISL, which are respectively used to express how protocol messages are encoded, and how interaction by transmitting these messages affects the states of participants in the protocol.

A framework was written that is capable of compiling APSL modules, and can provide a convenient representation of their contents that could be used for various applications. Two such applications have been implemented: automatic message parsers and serializers, and tools capable of carrying out a conformance test against a real system by solely sending particular messages and examining the responses.

A method was devised for evaluating the effectiveness of these testing strategies, and was applied when trying to use to whole system to test real implementations of the WebSocket and IMAP protocols.

While quite some improvements will still have to be made before this language, and the accompanying testing system, become a practical tool for conveniently testing protocol implementations, the initial results are promising. Furthermore, this approach is rather unique, and may provide an interesting subject within the realms of specification languages and automated testing.

9 Source Code and Resources

Grammars and more detailed documentation for the APSL language, source code for the compiler and tools, and test results can be found through the following URL:

`https://tinyurl.com/jtgbgwo`

References

[1] Postel, John, and J. Reynolds. "Instructions to RFC authors." (1997).

[2] Scott, Gregor D. "Guide for internet standards writers." (1998).

[3] Dick, Grune, and H. Ceriel. "Parsing Techniques, a Practical Guide." Technical Report, 1990.

[4] Johnson, Stephen C. "Yacc: Yet another compiler-compiler." Vol. 32. Murray Hill, NJ: Bell Laboratories, 1975.

[5] Bray, Tim, et al. "Extensible markup language (XML)." World Wide Web Consortium Recommendation REC-xml-19980210. (1998)

[6] Fallside, David C., and Priscilla Walmsley. "XML schema part 0: primer second edition." W3C recommendation (2004)

[7] Zyp, Kris. "A JSON media type for describing the structure and meaning of JSON documents." draft-zyp-json-schema-02 (work in progress) (2010).

[8] "Application fields of ASN.1." International Telecommunication Union.
http://www.itu.int/en/ITU-T/asn1/Pages/Application-fields-of-ASN-1.aspx

[9] "Abstract Syntax Notation One (ASN.1) Recommendations." International Telecommunication Union. http://www.itu.int/ITU-T/studygroups/com17/languages/

[10] Overell, Paul, and Dave Crocker. "Augmented BNF for syntax specifications: ABNF." (2008).

[11] "The Encoding control notation." International Telecommunication Union. http://www.itu.int/en/ITU-T/asn1/Pages/ecn.aspx

[12] "Protocol Buffers." Google Developers. https://developers.google.com/protocol-buffers/

[13] "Apache Thrift." Apache Foundation. https://thrift.apache.org/

[14] Pang, Ruoming, et al. "binpac: A yacc for writing application protocol parsers." Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. ACM, 2006.

[15] ITU-T, Recommendation Z., and Z. Recommendation. "100: specification and description language (SDL)." International Telecommunication Union (2000).

[16] Grammes, Rdiger, and Reinhard Gotzhein. "SDL Profiles - Formal Semantics and Tool Support." Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2007. 200-214.

[17] "A formal description technique based on an extended state transition model." ISO Standard IS9074, ISO: International Standards Organisation (1989).

[18] Bochmann, Gregor V., George Walter Gerber, and J-M. Serre. "Semi-automatic implementation of communication protocols." Software Engineering, IEEE Transactions on 9 (1987): 989-1000.

[19] Fecko, Mariusz A., et al. "A success story of formal description techniques: Estelle specification and test generation for MIL-STD 188-220." Computer Communications 23.12 (2000): 1196-1213.

[20] Bochmann, Gregor V., and Alexandre Petrenko. “Protocol testing: review of methods and relevance for software testing.” Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis. ACM, 1994.

[21] Lee, David, and Mihalis Yannakakis. “Principles and methods of testing finite state machines-a survey.” Proceedings of the IEEE 84.8 (1996): 1090-1123.

[22] Yannakakis, Mihalis, and David Lee. “Testing finite state machines: fault detection.” Journal of Computer and System Sciences 50.2 (1995): 209-227.

[23] Takanen, Ari. “Fuzzing: the Past, the Present and the Future.” Actes du 7ème symposium sur la sécurité des technologies de l’information et des communications (SSTIC). 2009.

[24] “Peach Fuzzer.” `http://www.peachfuzzer.com/`

[25] Claessen, Koen, and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs.” Acm sigplan notices 46.4 (2011): 53-64.

[26] Beurdouche, Benjamin, et al. “A messy state of the union: Taming the composite state machines of TLS.” IEEE Symposium on Security and Privacy. IEEE. 2015.

[27] Martens, Wim, Frank Neven, and Thomas Schwentick. “Complexity of decision problems for simple regular expressions.” Mathematical Foundations of Computer Science 2004. Springer Berlin Heidelberg, 2004. 889-900.

[28] Hewitt, Carl, Peter Bishop, and Richard Steiger. “A universal modular actor formalism for artificial intelligence.” Proceedings of the 3rd international joint conference on Artificial intelligence. Morgan Kaufmann Publishers Inc., 1973.

[29] Brand, Daniel, and Pitro Zafiropulo. “On communicating finite-state machines.” Journal of the ACM (JACM) 30.2 (1983): 323-342.

[30] Cardoso, Janette, and Heloisa Camargo, eds. “Fuzziness in Petri nets.” Vol. 22. Springer Science & Business Media, 1998.

[31] Holzmann, Gerard J. “The SPIN model checker: Primer and reference manual.” Vol. 1003. Reading: Addison-Wesley, 2004.

[32] Van Deursen, Arie, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography.” Sigplan Notices 35.6 (2000): 26-36.

[33] Hudak, Paul. "Building domain-specific embedded languages." ACM Computing Surveys (CSUR) 28.4es (1996): 196.

[34] Fielding, R., et al. "Hypertext Transfer Protocol - HTTP/1.1" (1999).

[35] Dale, Nell, and Henry M. Walker. "Abstract data types: specifications, implementations, and applications." Jones & Bartlett Learning, 1996.

[36] Pellauer, Michael, Markus Forsberg, and Aarne Ranta. "BNF Converter: Multilingual front-end generation from labelled BNF grammars." Technical report, Computing Science at Chalmers University of Technology and Gothenburg University, 2004.

[37] Lennart Kolmodin. "binary-bits: Bit parsing/writing on top of binary." `http://hackage.haskell.org/package/binary-bits`

[38] Tretmans, Gerrit Jan. "A formal approach to conformance testing." (1992).

[39] Prasetya, IS Wishnu B. "T3, a combinator-based random testing tool for Java: Benchmarking." Future Internet Testing. Springer International Publishing, 2013. 101-110.

[40] Prasetya, I. S. "T3i: a tool for generating and querying test suites for Java." Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015.

[41] "Utrecht University Attribute Grammar Compiler." `http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem`

[42] Middelkoop, Arie, Alexander B. Elyasov, and Wishnu Prasetya. "Functional Instrumentation of ActionScript Programs with Asil." IFL. 2011.

[43] Afzal, Wasif, Richard Torkar, and Robert Feldt. "A systematic review of search-based testing for non-functional system properties." Information and Software Technology 51.6 (2009): 957-976.

[44] Fraser, Gordon, and Andrea Arcuri. "Evosuite: On the challenges of test case generation in the real world." Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on. IEEE, 2013.

[45] Rueda, Urko, et al. "Unit testing tool competition: round four." Proceedings of the 9th International Workshop on Search-Based Software Testing. ACM, 2016.

[46] Nie, Changhai, and Hareton Leung. "A survey of combinatorial testing." ACM Computing Surveys (CSUR) 43.2 (2011): 11.

[47] Fette, I., and A. Melnikov. "RFC 6455: The websocket protocol." IETF, December (2011).

[48] "Autobahn: Open-source real-time framework for Web, Mobile & Internet of Things." http://autobahn.ws

[49] Crispin, M. "RFC 3501: Internet Message Access ProtocolVersion 4rev1 (2003)."

[50] "Courier IMAP." Double Precision, Inc. http://www.courier-mta.org/imap

[51] Sloane, Neil JA. "Covering arrays and intersecting codes." Journal of combinatorial designs 1.1 (1993): 51-63.

[52] Charlie Colbourn. "Covering Array Tables for t=2,3,4,5,6: Table for CAN(3,k,20) for k up to 10000." http://www.public.asu.edu/~ccolbou/src/tabby/3-20-ca.html