Apache Daffodil for Network Traffic Validation

Ben Cavanagh and Stephen Taylor
Dartmouth College, Hanover NH, USA

benjamin.y.cavanagh.27@dartmouth.edu
stephen.taylor@dartmouth.edu

Abstract: This paper explores the relative merit of Apache Daffodil for the automated generation of real-time network traffic validators associated with military vehicles. Daffodil is an "open-source implementation of the Data Format Description Language (DFDL)" which converts between natively formatted data and Extensible Markup Language (XML), JavaScript Object Notation (JSON) or other structures based on predefined schemas. Daffodil can also automatically generate executable parsers, and it is the efficacy and maturity of this facet of the technology for traffic validation that is our primary interest. Military vehicles have traditionally used compact protocols, such as Space Packet Protocol (SPP), MIL-STD-1553, or Controller Area Network bus (CAN bus), to enable real-time validation and limit malicious implants embedded in normal traffic. Over time, message traffic has grown increasingly complex due to increases in performance and use of commercial-off-the-shelf networking technologies leveraging Ethernet and TCP/IP protocols. Messages may now include mission-specific or recursively defined data embedded in the payload fields of variable-length packets. Consequently, it is valuable to assess Daffodil's ability to handle both existing formats and representative payload data within this emerging context. Here we directly compare DFDL schemas with equivalent grammars expressed in an extended version of Backus-Naur Form (xBNF) and Hammer to assess its expressive capability and succinctness. To assess the maturity of its validator generator, we compared Daffodil to the industry-standard Bison tool (for xBNF) and Hammer library. Though the comparison is based on numerous formats, here we present three examples: JSON numbers—a simple recursive data format; Micro Air Vehicle Link (MAVLink) messages—non-trivial byte-oriented traffic; and generic SPP messages—a bit-oriented, fixed-size format. Unfortunately, Daffodil's automated parser generation capabilities are currently neither mature nor robust. Daffodil is primarily useful for analysing fixed-length formats or variable-length components which follow descriptive header fields. Daffodil parsers can elegantly check compact bit-wise protocol formats, which is more difficult using Bison and some versions of Hammer. Daffodil is currently of limited utility in analysing recursive or non-prefixed formats such as JSON. Writing DFDL manually appears significantly more verbose and error-prone than both xBNF and Hammer.

Keywords: Apache Daffodil, Bison, Hammer, Parser generation, Traffic validation, Secure parsing

1. Introduction

Many organizations handle sensitive data and files relating to military missions, trade secrets, intellectual property, private personal data, and/or classified projects (NBAR, 2021). Traditionally, these organizations have been well advised to implement an air gap that physically isolates computers containing sensitive information from the Internet to protect against theft. Unfortunately, air gaps come with a substantial cost in productivity and assume that professional staff, with the expertise to handle sensitive information, are co-located. Consequently, individual air, space, and ground vehicles increasingly rely upon standard networking technologies to link embedded control systems with sensors, actuators, and human-machine interfaces through industry standard buses (e.g., CAN, J1939, MIL-STD-1553, USB) and communications interfaces (e.g., Gigabit Ethernet, OpenVPX, and PCIe). Network connected personal devices—phones, tablets, and laptops—are increasingly used to manage and interact with these systems. Though conceptually air-gapped, these systems are intermittently connected with military installations to provide mission parameters, or to effect maintenance and upgrades. Network boundary protections are used during these activities, but threat vectors including unintended connections, insiders, zero-day exploits, supply chain interdiction, and persistent implants can circumvent such protections (Kushner, 2013).

To combat these threats, byte-by-byte validation of message data can be implemented below the network stack—within the kernel, hypervisor, or underlying hardware—to ensure data is well-formed and contains reasonable values relative to the intended mission. Such validation makes use of *parsers*, which implement well-defined *grammars* representing the acceptable syntactic structure of messages. Traditionally, parsers match data to grammatical rules and generate parse trees, or structural representations. Validators must only establish that the input is consistent with the grammar and return a binary valid/invalid result. Beyond pure syntactic validation, more complex programs may also verify that data falls within reasonable threshold values, for example regarding the physical status of a vehicle within its intended mission. This paper explores the relative capabilities of Bison, Hammer, and Apache Daffodil with regards to the generation of validation parsers as part of a larger network traffic security toolchain.

1.1 Automated Parser Generation

Parser generators are tools that take a grammar as input and automatically generate a program that implements the associated parser. Here we consider three alternative approaches to automated parser generation—Bison (Levine, 2009), Hammer (Bratus et al, 2016), and Apache Daffodil (W3C, 2012)—to assess their relative strengths for network traffic validation. All three tools can operate on both textual (i.e. ASCII) and binary data. Though we have explored many grammars, we present observations here in the context of three formats that exemplify our findings: JavaScript Object Notation Numbers (JNUM) (ECMA, 2017), Micro Air Vehicle Link (MAVLink) (Dronecode Foundation, n.d.) protocol messages, and Space Packet Protocol (SPP) messages (CCSDS, 2020). JSON numbers involve recursively structured, byte-oriented data; MAVLink uses prefixed variable-length byte-oriented messages, and SPP messages use bit-oriented fields.

1.1.1 Bison

Bison is the most mature and widely used parser generator. It accepts context free grammars (CFGs) and primarily generates two classes of parser: generalized left-to-right (GLR) and the more restrictive look-ahead left-to-right (LALR). Input CFGs are expressed as a set of parsing rules in Backus-Naur Form (BNF) (Backus et al, 1960). Each rule expresses how a non-terminal symbol in the grammar is composed of terminal symbols such as ASCII characters (e.g. 'a'), binary values expressed in hexadecimal (e.g. '\xF5'), or other non-terminal symbols. Recursive rules can describe *sequences* of symbols, and multiple rules associated with the same non-terminal provide *choice* in how to parse a particular construct.

Though GLR parsers are more capable, LALR parsers are sufficient for validating a wide variety of communication protocols and file formats. Their simplicity allows them to be realized by a simple push-down automaton—a finite state machine employing a single stack to store symbols while parsing the input stream. The state machine relies on two core operations: shift—saving an input symbol to the stack—and reduce—the application of a grammar rule to detect a structure and reduce the symbols on the stack. The order in which shift and reduce operations are applied to the input is based on a collection of parsing tables derived from the grammar, usually referred to as Action/Goto tables (Aho, Sethi, and Ullman, 1988). These tables map the current state of the automaton to a next state based on the symbol read from the input stream.

Unfortunately, Bison's version of BNF cannot natively express common constructs in communication protocols such as strings (e.g. 'true'), wildcard values matching any input byte (often written *), fixed width data types (e.g. uint_64, int_16 etc.) in big- or little-endian formats, or ranges of values (e.g. ['a'-'f'] representing a character between 'a' and 'f' inclusive). Consequently, here we use an extended form of BNF (xBNF) to simplify each grammar using these additional notations (Taylor and Meise, 2021). The extended form is implemented as a preprocessor, written in Bison.

1.1.2 Hammer

Hammer is modern "parser combinator" library developed under the DARPA SafeDocs program. The library provides a collection of formally-defined base parsers and rules, designated with an *H_RULE()* construct, which are combined to build more complex parsers—all expressed in the C programming language. The more complex parser combinators *h_sequence()* and *h_choice()* allow sequences of parsers and choices between alternative parsers, respectively, to be composed. The resulting parsers are provably correct by construction. The Hammer library provides a collection of back ends that allow different parsing algorithms to be employed, including Packrat, providing recursive descent parsing, and LALR, providing bottom-up look-ahead parsing. Unfortunately, only the Packrat back end supports bit-oriented combinators, which can directly parse bit fields instead of having to pad each field to byte length through preprocessing. Further, the LALR back end does not support the little-endian combinators needed for MAVLink. As such, the Hammer versions of the JNUM parser used here make use of the LALR back end to match that of xBNF, while the MAVLink and SPP parsers use the Packrat back end.

1.1.3 DFDL and Apache Daffodil

The Data Format Description Language (DFDL) is a specification maintained by the Open Grid Forum for describing both textual and binary data formats. It is based on the W3C XML Schema Definition Language (XSD), which was created to "express syntactic, structural and value constraints" for text data structured or stored with XML. XSD schemas are made up of element declarations, which are assigned types and may also be assigned additional annotations. XSD includes many primitive types, of which DFDL includes only a subset. User-defined types may either be 'simple' or 'complex' and are created from these primitives. An *xs:simpleType* is created by modifying a primitive base type with additional annotations. The most common annotation here is an

xs:enumeration, which defines the terminal symbols which are valid for each type. The key differentiator for *xs:complexType* definitions is the presence of one or more *xs:sequence* or *xs:choice* groups. Sequence and choice groups are the fundamental structures used to build up schemas and are used similarly to Bison *sequences* (i.e. recursive rules) and *choices* (i.e. multiple rules) or Hammer *h_choice()* and *h_sequence()* combinators. In the DFDL schemas that follow, types are defined to mirror Bison non-terminal symbols and Hammer *H_RULE()* constructs. Each DFDL schema has a top-level element whose type is a complex type defining the entire grammar and is used as the starting point for the parser.

Apache Daffodil is an open-source implementation of DFDL, which provides support for data parsing, converting application-specific data formats to the DFDL-native Information Set (Infoset). The Daffodil library also provides support for 'unparsing,' or data serialization, from the Infoset to the application-specific format. Parsing and unparsing are performed by the library, with the input data and XSD schema taken as arguments for each routine at runtime. Daffodil also supports 'saving' schemas to a binary file for use in place of the XSD. Finally, Daffodil allows for the conversion of schemas into recursive descent parsers expressed in C. It is these generated C parsers that are of primary interest, as they allow parsing to be performed by a stand-alone executable, independent of the entire Daffodil library. However, Daffodil's C code generation currently supports only a subset of all features. Critically, type definitions which include elements of unspecified or expression-based length do not convert cleanly into C code.

1.1.4 IBM App Connect Enterprise

IBM App Connect Enterprise (ACE) is an enterprise integration software product which supports DFDL as part of its ability to facilitate data sharing and information flow (IBM Corp., 2025). IBM ACE was explored because its graphical user interface might have made schema writing simpler. Unfortunately, though perhaps less error prone than manual typing, our experience was that the ACE editor was too cumbersome to quickly develop schemas. Additionally, ACE does not support or focus on the generation of standalone validators, meaning that this product could only be used as an alternative schema editor in the context of a mission-ready toolchain.

2. JSON Numbers - Simple Recursive Format

The JavaScript Object Notation (JSON) format is a widely used data format used for storage and transmission. JSON numbers (JNUM) are one of the basic component types used to represent information and were selected to illustrate a simple recursive, byte-oriented data format. A JSON number (JNUM), such as 12.34e+56, is made up of an integer (the part preceding the decimal place, e.g. '12'), an optional fraction (the decimal point and following digits, e.g. '.34') and an optional exponent (an 'e' or 'E' followed by an optional sign and digits, e.g. 'e+56'). Note that JNUM requires the ability to handle input data of unspecified length, as each of the integer, fraction, and exponent fields may have any number of digits. Further, the format requires the ability to handle optional elements without the presence of a preceding header or 'flag' field.

2.1 xBNF

The recursive structure of xBNF makes the implementation of JNUM simple; the entire grammar, below, is just 8 lines. Non-terminal symbols are constructed to match the official specification, which directly specifies sequences and choices. The initial rule (line 1) provides a top level for the grammar (*jnum*) and designates a *sequence* of three non-terminals (i.e. integer, fraction, and exponent). Use of the *choice* symbol '|' (read as 'or') in the subsequent rules provides a shorthand for alternative rules to reduce the same non-terminal. For example, the *digit* rule (line 7) could equivalently be expressed as two rules that provide two ways to reduce the same non-terminal symbol *digit*. The end-of-input must occur when the *jnum* rule is successfully reduced, with no additional data on the input. Importantly, the recursive *digits* rule (line 6) allows the grammar to handle data of unspecified length; in this case *digits* is a sequence of one or more individual instances of a single *digit*. Also note that optional elements are specified simply by providing an *empty* choice (lines 3, 4, 5), by convention marked with a comment such as in the definition of a *sign*.

```
jnum: integer fraction exponent ;                              /* 1 */
integer: digit | onenine digits | '-' digit | '-' onenine digits ;   /* 2 */
fraction: /* empty */ | '.' digits ;                           /* 3 */
exponent: /* empty */ | 'E' sign digits | 'e' sign digits ;    /* 4 */
sign: /* empty */ | '+' | '-' ;                                /* 5 */
```

```
digits: digit | digits digit ;                    /* 6 */
digit: '0' | onenine ;                           /* 7 */
onenine: [ '1' - '9' ] ;                         /* 8 */
```

The ability for xBNF to concisely represent range expansions (line 8), as opposed to a manually created sequence of choice fields as in raw BNF, is also significant. As the target format becomes more complex, the manual expression of ranges quickly becomes unwieldy. The above grammar can be directly translated into a software parser using Bison for use at the application, kernel, or hypervisor levels of the software stack; alternatively it can be directly translated into a highly compact hardware circuit for use in a field programmable gate array (FPGA) or system-on-chip (SoC) device (Taylor, 2022 and Taylor, 2024).

2.2 Hammer

The Hammer JNUM implementation uses a top-level '*jnum_parser*' rule at the end of the parser. It is defined with an *h_sequence()* combinator containing a '*jnum*' parser, *h_end_p()* primitive (to designate where the end of input should occur), and *NULL* to indicate the end of the parameter sequence:

```
jnum_parser = h_sequence(jnum, h_end_p(), NULL);
```

In a similar manner to xBNF, the *jnum* rule is made up of smaller user-defined parsers that are assembled with further *h_sequence()* and *h_choice()* combinators:

```
H_RULE(digits, h_many1(digit));
H_RULE(integer, h_sequence(h_optional(h_ch('-')),
        h_choice(digit, h_sequence(onenine, digits, NULL), NULL), NULL));
H_RULE(fraction, h_sequence(h_ch('.'), digits, NULL));
H_RULE(exponent, h_sequence(h_choice(h_ch('E'), h_ch('e'), NULL),
        h_optional(sign), digits, NULL));
H_RULE(jnum, h_sequence(integer, h_optional(fraction),
        h_optional(exponent), NULL));
```

Note that because Hammer is written in C, lower-level elements are defined first. Hammer uses an explicit *h_optional()* combinator to denote optional symbols rather than the empty rules used by xBNF. Also note the *h_many1()* combinator is used in place of recursive rule definitions. At the lowest level, *H_RULE* parsers combine primitive parsers, such as characters (i.e. *h_ch()*) and ranges of characters (i.e. *h_ch_range()*) to allow symbol enumeration. Hammer is more verbose than xBNF (Taylor, 2022), in part because it requires more setup notation. This parser required 21 lines of code; the setup and more basic elements are omitted here for brevity.

2.3 DFDL

The DFDL schema for JNUM was developed to match xBNF symbol-by-symbol build-up with a top-level *number* non-terminal. The DFDL equivalent is the *JSON_number* global element, which has user-defined complex type *number_t*. This *number_t* then contains an *xs:sequence* and matches the integer, fraction, exponent ordering above:

```
<xs:complexType name="number_t">
 <xs:sequence>
  <xs:element name="integer" type="integer_t"/>
  <xs:element name="fraction" type="fraction_t" minOccurs="0" maxOccurs="1"/>
  <xs:element name="exponent" type="exponent_t" minOccurs="0" maxOccurs="1"/>
 </xs:sequence>
</xs:complexType>
```

DFDL marks optional components in the grammar through the *minOccurs="0"* annotation, in contrast to the xBNF choice of an *empty* element. Within the *xs:sequence*, each of the local elements itself is assigned a complex type which contains further sequence or choice fields. Simple types are defined in terms of primitives and an enumerated set of legal values. For example, the JNUM schema contains a *zero_t* type:

```
<xs:simpleType name="zero_t" dfdl:lengthKind="explicit" dfdl:length="1">
 <xs:restriction base="xs:unsignedByte">
  <xs:enumeration value="&#x30;"/>
 </xs:restriction>
</xs:simpleType>
```

It is clear from the code snippets above that the DFDL grammar definition is significantly more verbose than either xBNF or Hammer. In total, 153 lines of code are required. DFDL supports more compact ways to parse messages—JNUM, for example, can be validated with an *xs:simpleType* element based on a string primitive as a single line of DFDL regex (Beckerle, 2021 pp 196). However, these other methods do not actually break up the incoming message into individual elements nor represent its internal structure for conversion into a validator.

Critically, JNUM's structure requires parse-time decision making based on the value of the token currently being consumed. This requires the *dfdl:discriminator* annotation, which is currently unsupported in the C code generator. As such, an executable JNUM parser could not be generated using Daffodil.

2.4 Summary of Results

Grammar or Schema	Lines of Code	User-Defined Non-terminals	Executable Size (KB)	Fuzz Testing Execution Time (s, n= 42)
xBNF	8	8	21.6	0.088
Hammer	21	8	193.2	0.128
DFDL	153	12	—	—

3. MAVLink Protocol - Variable Length Format

The Micro Air Vehicle Link (MAVLink) Protocol is a communications protocol used with small unmanned aerial vehicles. It supports the transmission of information regarding a drone's status as well as more complex 'microservice' protocols, including a heartbeat and mission protocol. MAVLink currently has both a version 1 and version 2; MAVLink v2 messages have longer headers, may add a signature field at the end of the message, and require the truncation of empty bytes at the end of the message. This last requirement dramatically increases the number of possible message states in MAVLink v2, as each message type may now carry a variable length payload depending on whether fields have been truncated.

Representations of three message types, namely the HEARTBEAT, SYS_STATUS, and VFR_HUD messages for both MAVLink v1 and v2, were created in xBNF, Hammer, and DFDL to explore the capability of each to parse non-delimited formats and verify bounded or enumerated values.

3.1 xBNF

In xBNF, non-terminals are again built up from combinations of *sequence* and *choice* constructs. The context-free nature of xBNF means that a non-terminal must be created for every possible combination of message options. For example, to support all three message types in MAVLink v1, v2 signed, and v2 unsigned, each of the nine possible options must be represented separately. The following code snippet showcases the top-level non-terminals which guide these decisions:

```
P : message ;
/* Message is either MAVLink v1, v2 signed, or v2 unsigned */
message : msg_v1 | msg_v2 ;
msg_v2 : msg_sg | msg_us ;
/* HEARTBEAT, SYS_STATUS, or VFR_HUD message */
msg_v1 : stx_v1 msg_v1_opts crc ;
```

```
msg_sg : stx_v2 msg_sg_opts crc sig ;
msg_us : stx_v2 msg_us_opts crc ;
msg_v1_opts : v1_hb | v1_ss | v1_hud ;
msg_sg_opts : sg_hb | sg_ss | sg_hud ;
msg_us_opts : us_hb | us_ss | us_hud ;
```

The compact syntax of xBNF makes the actual representation of non-terminals simple, but as the number of choices and their options grows, the requirement to independently represent each combination quickly grows the size of the grammar. Since MAVLink v2 payloads truncate empty trailing bytes, for example, a payload whose defined fields are N bytes long may in actuality be transmitted at sizes from 1 to N bytes. This requires 2N non-terminals just to represent the possible combinations of the message being signed/unsigned and having a payload of a certain length. For the MAVLink 3-message grammar, xBNF required 337 lines of code and the definition of 251 non-terminals to achieve a readable grammar that correlates well with the specification.

3.2 Hammer

The Hammer MAVLink parser closely mirrors the xBNF implementation through its use of *sequence* and *choice* combinators. Below is the same representation of top-level MAVLink structures in Hammer:

```
H_RULE(msg_v1, h_sequence(stx_v1, h_choice(v1_hb, v1_ss, v1_hud, NULL), crc, NULL));
H_RULE(msg_sg, h_sequence(stx_v2, h_choice(sg_hb, sg_ss, sg_hud, NULL), crc, sig, NULL));
H_RULE(msg_us, h_sequence(stx_v2, h_choice(us_hb, us_ss, us_hud, NULL), crc, NULL));
H_RULE(message, h_choice(msg_v1, msg_sg, msg_us, NULL));
mav_3_enum_parser = h_sequence(message, h_end_p(), NULL);
return(mav_3_enum_parser);
```

Again, the use of only sequence, choice, and range expansion combinators means that every possible combination of choices must be assigned its own rule. In the MAVLink example, Hammer is not significantly more verbose than xBNF: the parser required 372 lines of code and 247 *H_RULE()* non-terminal definitions.

3.3 DFDL

Since MAVLink messages have fixed-length headers which describe variable-length payload fields, it was possible to use Daffodil's C parser generator and create a standalone MAVLink parser. The implementation is heavily dependent on the *dfdl:occursCountKind="expression"* annotation, which allows the parser to refer back to previous message fields to guide parsing decisions. Since MAVLink messages are flagged with a start-of-text (STX) field indicating the protocol version (0xFE indicates MAVLink v1, 0xFD indicates MAVLink v2), Daffodil C's look-ahead capability is sufficient to guide parsing:

```
<xs:complexType name="message_t">
 <xs:sequence>
  <xs:element name="stx" type="stx_t"/>
  <xs:sequence>
   <xs:element name="M1" type="M1_t"
    minOccurs="0" maxOccurs="1"
    dfdl:occursCountKind="expression"
    dfdl:occursCount="{ if (../stx eq xs:hexBinary('FE')) then 1 else 0 }" />
   <xs:element name="M2" type="M2_t"
    minOccurs="0" maxOccurs="1"
    dfdl:occursCountKind="expression"
    dfdl:occursCount="{ if (../stx eq xs:hexBinary('FD')) then 1 else 0 }" />
```

```
        </xs:sequence>
      </xs:sequence>
    </xs:complexType>
```

This snippet illustrates the choice between a v1 or v2 message based on the deterministic parsing of the STX header field with an if-statement annotation. Similar annotations were used to verify the length of message payloads, check enumerated values associated with each message type, and determine the presence of a v2 signature field. A key benefit of the ability to refer to previous fields is that types representing separate choices can remain independent. For example, v2 SYS_STATUS messages may either be signed or unsigned and may also have payloads ranging from 1 to 31 bytes in length. In xBNF, a non-terminal must be created for each combination of signedness and length value. In DFDL, representations of these choices may be defined independently, reducing the total number of types. For a more complicated format with many more choices, this could imply a large reduction in the overall size of the schema. The DFDL MAVLink 3-message schema required only 110 user-defined non-terminal types. However, because of the verbosity of DFDL's syntax, the implementation was still much longer than either the xBNF or Hammer grammars at 906 lines of code.

The expressions which determine the value of the *dfdl:occursCount* annotations are written in the DFDL expression language, which is a subset of XML Path Language 2.0 (XPath 2.0). Currently, the Daffodil C generator does not fully support this behaviour: schemas which refer back to a 'grandparent' or higher elements (meaning they require two or more '*../*' in the expression) do not get converted correctly. There is also no support for embedded if-statements for input-based decision making. Consequently, we had to make modifications to the Scala generator back end: The issue with referencing grandparent elements was resolved by correcting the counting of '*../*' constructs prepended to an expression, while support for if-statements was added via a text-processing function.

3.4 Summary of Results

Grammar or Schema	Lines of Code	User-Defined Non-terminals	Executable Size (KB)	Fuzz Testing Execution Time (s, n= 54243)
xBNF	337	251	247	109.8
Hammer	372	247	215	113.6
DFDL	906	110	819	115.2

4. Space Packet Protocol - Bitwise Format

The Consultative Committee for Space Data Systems (CCSDS) Space Packet Protocol (SPP) (CCSDS, 2020) was chosen to test the capabilities of each library to parse its big-endian bit-oriented fields. The implementation of xBNF and Bison used here is byte-based and thus does not support parsing of natural SPP packets without a preprocessor that converts bit-fields into bytes. As such, only the Hammer Packrat back end and Daffodil were compared. A further challenge presented by SPP is the fact that its length field contains a count whose value is one less than the number of following octets of data. Practically, this requires the ability to perform arithmetic on parsed fields (i.e. add one), as the data field can contain up to 65,536 octets.

SPP messages are highly variable and mission-specific, and the official standard specifies little beyond high-level packet structures. Consequently, the example grammars below validate only the standard primary header fields directly. Any following secondary header or user data field (UDF) must contain an integral number of octets which matches the Packet Data Length field, as specified in the CCSDS standard. As the secondary header is currently unspecified (SANA 2020), it is assumed that only one of a secondary header or UDF is present. The single exception is the 'idle' packet, which contains a specific 'idle data' UDF. For the grammars below, the two-octet sequence '0xAA' acts as example idle data and is verified bit-by-bit.

4.1 Hammer

Bit-oriented SPP required the use of two additional features in Hammer: the *h_attr_bool()* combinator, which allowed the validation of enumerated bit fields, as well as the *h_action()* combinator, which embeds user-defined functions to correct for the off-by-one count error described above. In both cases, ad-hoc user-written C functions were required. The *h_attr_bool()* combinator takes the bitwise parse result from the *h_bits()* parser and passes it to a user-defined function which checks its value. For the SPP parser, two functions were created simply to verify that a bit is a '1' or '0' respectively. A third function validates the 11-bit APID field which denotes

an idle packet. Note this value validation was previously accomplished by the *h_ch()* combinator when fields were byte oriented.

The *h_action()* combinator works similarly, but instead of changing the validator result (success/failure) it modifies the parsed value. In this case, the parsed count field had to be incremented for use in verifying the number of octets in the UDF. The requirement to write out entire functions to accommodate these more complex behaviours made the grammar more verbose, and 97 lines of code with 19 non-terminals were required.

4.2 DFDL

In DFDL, a grammar can be made bit-oriented simply with the *lengthUnits="bits"* format annotation in the preamble, and as such the validation of enumerated values can be performed straightforwardly. Further, XPath 2.0 expressions make it possible to add 1 to the primary header length field to count payload octets:

dfdl:occursCount="{ ../../../prim_hdr/len + 1 }"

The smaller modifications required to make these more complex parsing decisions possible meant that DFDL was simpler to write: the grammar required 124 lines of code and 8 user-defined types; unfortunately, the resulting executable code was 175 times larger and less than half as quick.

4.3 Summary of Results

Grammar or Schema	Lines of Code	User-Defined Non-terminals	Executable Size (KB)	Fuzz Testing Execution Time (s, n=190)
Hammer	97	19	199	0.375
DFDL	124	8	35083	0.901

5. Conclusion

Daffodil was found to be consistently more verbose, larger in executable size, and slower than either xBNF or Hammer. While Daffodil remains a capable data conversion library, its verbosity and limited support for the generation of standalone validators makes it currently unsuitable for standalone mission-ready traffic validation. Its inability to generate C parsers that process recursively defined formats is a more minor limitation given the prevalence of prefixed variable-length formats. However, its verbosity compared to xBNF and Hammer makes it unwieldy to express schemas; currently the Apache version does not support the automatic conversion of XPath 2.0 expressions. Existing support for parse-time decisions more generally, such as through the *dfdl:choiceDispatchKey* or *dfdl:discriminator* annotations, is limited and impacts the formats which generated C parsers can validate. Finally, Daffodil's recursive descent back end is likely to make its parsers less efficient in FPGA resources if implemented in hardware.

Daffodil parsers do demonstrate the potential for simplifying the grammars used to describe prefixed formats through the ability to refer to previously parsed header fields containing descriptive information. Compared to xBNF and Hammer, which required 251 and 247 non-terminals respectively, Daffodil required the definition of just 110 user-defined types for 3-message MAVLink. A more concise description language that supports a supplement to CFGs, such as an attribute grammar, may allow the recollection of such descriptive fields and could significantly simplify grammars. If such an implementation also allowed for table-driven parsing, it may be a good alternative for secure traffic validation.

Ethics Declaration: Ethical clearance was not needed for this research.

AI Declaration: AI tools were not used in the creation of this paper.

References

Aho, A.V., Sethi, R., and Ullman, J.D. (1988) Compilers, Addison Wesley.

Backus, J.W. et al (1960) "Report on the Algorithmic Language ALGOL 60", *Commun. ACM*.

Beckerle, M.J. and Hanson, S.M. (2021) "Data Format Description Language (DFDL) v1.0 Specification", [online], Open Grid Forum, https://ogf.org/documents/GFD.240.pdf.

Bratus, S. et al (2016) “Implementing a vertically hardened DNP3 control stack for power applications”, ICSS'16 ACM Proceedings of the 2nd Annual Industrial Control System Security Workshop, pp 1-9. (c.f. https://gitlab.specialcircumstanc.es/hammer/hammer).

Consultative Committee for Space Data Systems (2020), "Space Packet Protocol", CCSDS 133.0-B-2, [online], https://ccsds.org/publications/bluebooks/entry/3264/.

Dronecode Project. (n.d.) "MAVLink Developer Guide", [online], The Linux Foundation, https://mavlink.io/en/
ECMA International (2017) "The JSON Data Interchange Syntax", [online], https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf .
Kushner, D. (2013) “The Real Story of STUXNET”, IEEE Spectrum.
IBM Corporation. (2025) "IBM App Connect Enterprise", [online], IBM Documentation, https://www.ibm.com/docs/en/app-connect/13.0.x?topic=editors-dfdl-schema-editor.
Levine, J. (2009) *flex & bison*, O'Reilly, Sebastopol.
NBAR (2021) -- The National Bureau of Asian Research, The IP Commission Report, 2013, updated 2021.
Space Assigned Numbers Authority (2020) "Space Packet Protocol Secondary Header Format Document", [online], https://sanaregistry.org/r/space_packet_protocol_secondary_header_format_document/.
Taylor, S. and Meise, J. (2021) "xBNF -- A preprocessor, written in Bison, that extends Bison BNF", [online], Thayer School of Engineering, https://github.com/lvln/thayer_parsers/blob/main/xbnf/README.md.
Taylor, S. et al (2022) "Automatic Construction of Hardware Traffic Validators", *Proceedings of the 21st European Conference on Cyber Warfare and Security* Vol 21, No. 1, pp 60-69.
Taylor, S. and Pope, G. "Hardware Sequence Combinators", *Proceedings of the 19th International Conference on Cyber Warfare and Security* Vol 19, No. 1, pp 358-365.
World Wide Web Consortium. (2012) "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures", [online], W3C, https://www.w3.org/TR/xmlschema11-1/#intro-purpose.