Workgroup: Network Working Group
Internet-Draft:
draft-petithuguenin-computerate-specifying-03
Published: 9 March 2020
Intended Status: Experimental
Expires: 10 September 2020
Authors: M. Petit-Huguenin
Impedance Mismatch LLC

The Computerate Specifying Paradigm

Abstract

This document specifies a paradigm named Computerate Specifying, designed to simultaneously document and formally specify communication protocols. This paradigm can be applied to any document produced by any Standard Developing Organization (SDO), but this document targets specifically documents produced by the IETF.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 10 September 2020.

Table of Contents

1. Introduction

If, as the unofficial IETF motto states, we believe that "running code" is an important part of the feedback provided to the standardization process, then as per the Curry-Howard equivalence [Curry-Howard] (that states that code and mathematical proofs are the same), we ought to also believe that "verified proof" is an equally important part of that feedback. A verified proof is a mathematical proof of a logical proposition that was mechanically verified by a computer, as opposed to just peer-reviewed.

The "Experiences with Protocol Description" paper from Pamela Zave [Zave] gives three conclusions about the usage of formal specifications for a protocol standard. The first conclusion states that informal methods (i.e. the absence of verified proofs) are inadequate for widely used protocols. This document is based on the assumption that this conclusion is correct, so its validity will not be discussed further.

The second conclusion states that formal specifications are useful even if they fall short of the "gold standard" of a complete formal specification. We will show that a formal specification can be incrementally added to a standard.

The third conclusion from Zave's paper states that the normative English language should be paraphrasing the formal specification. The difficulty here is that to be able to keep the formal specification and the normative language synchronized at all times, these two should be kept as physically close as possible to each other.

To do that we introduce the concept of "Computerate Specifying" (note that Computerate is a British English word). "Computerate Specifying" is a play on "Literate Computing", itself a play on "Structured Computing" (see [Knuth92] page 99). In the same way that Literate Programming enriches code by interspersing it with its own

documentation, Computerate Specifying enriches a standard specification by interspersing it with code (or with proofs, as they are the same thing), making it a computerate specification.

Note that computerate specifying is not specific to the IETF, just like literate computing is not restricted to the combination of Tex and Pascal described in Knuth's paper. What this document describes is a specific instance of computerate specifying that combines [AsciiDoc] as formatting language and [Idris] as programming language with the goal of formally specifying IETF protocols.

2. Overview of Operations

Nowadays specifications at the IETF are written in a format named xml2rfc v3 [RFC7991] but unfortunately making that format "Computerable" is not trivial, mostly because there is no simple solution to mix code and XML together in the same file. Instead, we chose the AsciiDoc format as the basis for computerate specifications as it permits the generation of specifications in the xmlrfc v3 format (among other formats) and also because it can be enriched with code in the same file.

[I-D.ribose-asciirfc] describes a backend for the [Asciidoctor] tool that converts an AsciiDoc document into an xmlrfc3 document. The AsciiRFC document states various reasons why AsciiDoc is a superior format for the purpose of writing standards, so we will not discuss these further. Note that the same team developed Asciidoctor backends for other Standards Developing Organizations (SDO) [Metanorma], making it easy to develop computerate specifications targeting the standards developed by these SDOs.

The code in a computerate specification uses the programming language Idris in literate programming [Literate] mode using the Bird-style, by having each line of code starting with a ">" mark in the first column.

That same symbol was also used by AsciiDoc as an alternate way of defining a blockquote [Blockquotes] way which is no longer available in a computerate specification. Bird-style code will simply not appear in the rendered document.

The result of Idris code execution can be inserted inside the document part by putting that code fragment in the document between the "{`" string and the "`}" string.

A computerate specification is processed by an Asciidoctor preprocessor that does the following:

1. Loads the whole document as an Idris program, including importing modules.

2. For each instance of an inline code fragment, evaluates that fragment and replace it (including the delimiters) by the result of that evaluation.

3. Continues with the normal processing of the modified document.

For instance the following computerate specification fragment taken from the computerate specification of STUNbis

```
<CODE BEGINS>
> rto : Int
> rto = 500
>
> rc : Nat
> rc = 7
>
> rm : Int
> rm = 16
>
> -- A stream of transmission times
> transmissions : Int -> Int -> Stream Int
> transmissions value rto = value :: transmissions (value + rto)
>   (rto * 2)
>
> -- Returns a specific transmission time
> transmission : Int -> Nat -> Int
> transmission timeout i = index i $ transmissions 0 timeout
>
> a1 : String
> a1 = show rto
>
> a2 : String
> a2 = concat (take (rc - 1) (map (\t => show t ++ " ms, ")
>   (transmissions 0 rto))) ++ "and " ++ show (transmission rto
>     (rc - 1)) ++ " ms"
>
> a3 : String
> a3 = show $ transmission rto (rc - 1) + rto * rm
For example, assuming an RTO of {`a1`}ms, requests would be sent at
times {`a2`}.
If the client has not received a response after {`a3`} ms, the
client will consider the transaction to have timed out.
<CODE ENDS>
```

Figure 1

is rendered as

" For example, assuming an RTO of 500ms, requests would be sent at times 0 ms, 500 ms, 1500 ms, 3500 ms, 7500 ms, 15500 ms, and 31500 ms. If the client has not received a response after 39500 ms, the client will consider the transaction to have timed out."

Figure 2

Appendix A explains how to install the command line tools to process a computerate specification.

The Idris programming language has been chosen because its type system supports dependent and linear types, and that type system is the language in which formal specifications are written.

Following Zave's second conclusion, a computerate specification does not have to be about just formally specifying a protocol and proving properties about it. There is a whole spectrum of formalism that can be introduced in a specification, and we will present it in the remaining sections by increasing order of complexity. Note that because the formal language is a programming language, these usages are not exhaustive, and plenty of other usages can and will be found after the publication of this document.

2.1. Libraries

A computerate specification does not disappear as soon the standard it describes is published. Quite the opposite, each specification is designed to be used as an Idris module that can be imported in subsequent specifications, reducing over time the amount of code that needs to be written. At the difference of an RFC that is immutable after publication, the code in a specification will be improved over time, especially as new properties are proved or disproved. The latter will happen when a bug is discovered in a specification and a proof of negation is added to the specification, paving the way to a revision of the standard.

This document is itself a computerate specification that contains data types and functions that can be reused in future specifications, and as a whole can be considered as the standard library for computerate specifying.

For convenience, each public computerate specification, including the one behind this document, will be made available as an individual git repository. Appendix B explains how to gain access to these computerate specifications.

2.2. Retrofitting Specifications

RFCs, Internet-Drafts and standard documents published by other SDOs did not start their life as computerate specifications, so to be able to use them as Idris modules they will need to be progressively retrofitted. This is done by converting the documents into an AsciiDoc documents and then enriching them with code, in the same way that would have been done if the standard was developed directly as a computerate specification.

Converting the whole document in AsciiDoc and enriching it with code, instead of just maintaining a library of code, seems a waste of resources. The reason for doing so is to be able to verify that the rendered text is equivalent to the original standard, which will validate the examples and formal languages.

Retrofitted specifications will also be made available as individual git repositories as they are converted.

Because the IETF Trust does not permit modifying an RFC as a whole (except for translation purposes), a retrofitted RFC uses transclusion, a mechanism that includes parts of a separate document at runtime. This way, a retrofitted RFC is distributed as two separate files, the original RFC in text form, and a computerate specification that contains only code and transclusions.

Transclusion is a special form of AsciiDoc include that takes a range of lines as parameters:

```
include::rfc5234.txt[lines=26..35]
```

Figure 3

Here the include macro will be replaced by the content of lines 26 to 35 (included) of RFC 5234.

The "sub" parameter permits modifying the copied content according to a regular expression. For instance the following converts references into the AsciiDoc format:

```
include::rfc5234.txt[lines=121..131,sub="/\[([^\]])\]/<<\1>>/"]
```

Figure 4

In the following example, the text is converted into a note:

```
include::rfc5234.txt[lines=151,sub="/^.*$/NOTE: \0/"]
```

Figure 5

2.3. Implementation-Oriented Standards

After spending a few years implementing standard communication protocols, it becomes quite obvious that not all standards are meant to be directly converted into code. In most case an expert in both communication protocols and software development has to rearrange a standard and its set of dependencies into a form that can be implemented.

One sure sign that a standard has first to be rearranged is that the information needed to implement one single network element are spread all over the standard. For instance if for implementing the client side of a client-server protocol one has to collect information from most of the normative sections then that standard is not directly ready for implementation and requires first to put together all these pieces in a convenient form.

On the other hand some standards have been structured in a way that matches the workflow of a software implementer. [RFC8489] and [RFC8656] are examples of standards that are meant to be easily implemented.

Assuming that a standard is meant to be implemented, it follows that it should be a goal to publish it in a form that makes implementations easier on the software developers. On the other hand, writing a complete implementation as part of the development of an standard is a difficult task, especially as a standard will change over time during its development.

Because of the constraint that in a computerate specification the specification should be as close as possible to the equivalent normative text, standards developed using that technique tends to be naturally ready for implementation. This contrasts with the difficulties encounters when retrofitting an existing standard as a computerate specification, where trying to keep the specification close to the text is especially challenging.

2.4. Revision of Standards

Standards evolve, but because RFCs are immutable, revisions for a standard are done by publishing new RFCs.

The matching computerate specifications need to reflect that relationship by extending the data type of syntax and semantics in the new version, instead of recreating new data types from scratch. There are two diametrically opposed directions when extending a type:

* The new standard is adding constraints. This is done by indexing the new type over the old type.

*The new standard is removing constraints. This is done by defining the new type as a sum type, with one of the alternatives being the old type.

NOTE: This is correct in theory, but in practice creating new specifications from old ones as described above is not very convenient. Maybe an alternate solution is to define the new specifications from scratch, and use an isomorphism proof to precisely define the differences between the two. An Idris elaboration script may permit duplicating a type and modifying it without having to manually copy it.

2.5. Content of a Computerate Specification

Communication protocol specifications are generally split in two distinct parts, syntax (the data layout of the messages exchanged) and semantics (the rules that drive the exchange of messages).

Section 3 will discuss in detail the application of computerate specifying to syntax descriptions, and Section 4 will be about specifying semantics.

3. Syntax

The syntax of a communication protocol determines how data is laid out before it is sent over a communication link. Generally the syntax is described only in the context of the layer that this particular protocol is operating at, e.g. an application protocol syntax only describes the data as sent over UDP or TCP, not over Ethernet or Wi-Fi.

Syntaxes can generally be split into two broad categories, binary and text, and generally a protocol syntax falls completely into one of these two categories.

Syntax descriptions can be formalized for at least three reasons that will be presented in the following sections.

3.1. Syntax Examples

Examples in protocol documentation are frequently incorrect, which should not be that much of an issue but for the fact that most developers do not read the normative text when an example is available. See Appendix D for statistics about the frequency of incorrect examples in RFC errata.

Moving the examples into appendices or adding caution notices may show limited success in preventing that problem.

So ensuring that examples match the normative text seems like a good starting point for a computerate specification. This is done by having the possibility of adding the result of a computation directly inside the document. If that computation is done from a type that is (physically and conceptually) close to the normative text, then we gain some level of assurance that both the normative text and the derived examples will match. Note that examples can be inserted in the document as whole blocks of text, or as inline text.

Appendix B.2.1 showcases the conversion of the examples in [RFC5234].

3.1.1. Data Type

The first step is to define an Idris type that completely defines the layout of the messages exchanged. By "completely define" we mean that the type checker will prevent creating any invalid value of this type. That ensures that all values are correct by construction.

E.g. here is the definition of a DNS label per [RFC1034]:

```
<CODE STARTS>
> data PartialLabel' : List Char -> Type where
>   Empty : PartialLabel' []
>   More : (c : Char) -> (prf1 : isAlphaNum c || c == '-' = True) ->
>     PartialLabel' s -> (prf2 : length s < 61 = True) ->
>     PartialLabel' (c :: s)
>
> data Label' : List Char -> Type where
>   One : (c : Char) -> (prf1 : isAlpha c = True) -> Label' [c]
>   Many : (begin : Char) -> (prf1 : isAlpha begin = True) ->
>     (middle : PartialLabel' xs) ->
>     (end : Char) -> (prf2 : isAlphaNum end = True) ->
>     Label' ([begin] ++ xs ++ [end])
>
> data Label : {a : Type} -> a -> Type where
>   MkLabel : {xs : String} -> Label' (unpack xs) -> Label xs
<CODE ENDS>
```

Figure 6

3.1.2. Serializer

The second step is writing a serializer from that type into the wire representation. For a text format, it is done by implementing the Show interface:

```
<CODE STARTS>
> Show (Label xs) where
>   show _ = xs
<CODE ENDS>
```

Figure 7

NOTE: Define binary serializer.

3.1.3. Presentation Format

Instead of directly generating character strings, the serializer will use as target a dependent type that formalizes the AsciiDoc language. This will permit to know the context in which a character string will be subsequently generated and to correctly escape special characters in that string.

Using an intermediary type will also permit to correctly generate AsciiDoc that can generate an xmlrfc 3 file that supports both text and graphical versions of a figure. This will be done by having AsciiDoc blocks converted into <artwork> elements that contains both the 72 column formatted text and an equivalent SVG file, even for code source (instead of using the <sourcecode> element).

3.2. Designing a Data Type

Builtin data types in Idris are convenient but as a general rule should not be used in the design of data types meant to be used in the context of computerate specifications. Builtin types are abstract data types so most basic proofs about them have to be axioms, making it difficult to reason about them, which does not constitute a solid ground on which to build a verifiable system.

The builtin types in Idris are Int, Integer, Double, Char and String.

On the other hand user-defined data types like Nat, Fin, Dec (a proof carrying variant of Bool), Maybe, Either, List, Vect, etc. have already plenty of proofs that can be reused, so should be used in designing data types for computerate specifications.

Designing a user-defined data type is done by combining five basic types:

*Product type: This is a type built from concatenation of types, similar to Java's class or C's struct.

*Sum type: This is a type built from alternative types. It can be simulated with inheritance in Java, or a combination of struct and union in C. A Java enum is a very limited form of Sum type.

*Exponential type: This is the type of a total function (which implies free of side-effect and use of external variables).

*Pi type: This is a product type (or a exponential type) where the type depends on a value. This is equivalent to the universal quantifier.

*Sigma type: This is a sum type where the type depends on a value. This is equivalent to the existential quantifier.

In addition the underlying computerate specification of this document defines a set of data types that are suited for computerate specifications:

*The Bitvector type describes a fixed number of bits that can be manipulated together. The library contains functions to manipulate bitvectors that are inspired by the language in [Brinkmann02].

*The Unsigned type describes an unsigned integer data type that is built on top on the Bitvector library. The library contains ordering and arithmetic operators for unsigned numbers.

*The Signed type describes a signed integer data type that is built on top on the Bitvector library. The library contains ordering and arithmetic operators for signed numbers.

NOTE: Other RFCs are in the process of been retrofitted to implement data types built on top of the Bitvector type. This is the case for IPv4 and IPv6 address, MAC address, UUID, Unicode and many more. A future version of this document will add the IdrisDoc rendering of these libraries as appendices.

3.3. Formal Syntax Language

Some specifications use a formal language to describe the data layout. One shared property of these languages is that they cannot always formalize all the constraints of a specific data layout, so they have to be enriched with comments. One consequence of this is that they cannot be used as a replacement for the Idris data type described in Section 3.1.1, data type that is purposely complete.

The following sections describe how these formal languages have been or will be themselves formalized with the goal of using them in computerate specifications.

3.3.1. Augmented BNF (ABNF)

Augmented Backus-Naur Form [RFC5234] (ABNF) is a formal language used to describe a text based data layout.

The [RFC5234] document has been retrofitted as a computerate specification to provide an internal Domain Specific Language (DSL) that permits specifying an ABNF for a specification. The encoding of an example from Section 2.3 of [RFC5234] looks like this:

```
<CODE BEGINS>
> rulename : Rule
> rulename = "rulename" `Eq` (Concat (TermDec 97 []) (TermDec 98 [])
>   [TermDec 99 []])
<CODE ENDS>
```

Figure 8

A serializer, also defined in the same specification, permits converting that description into proper ABNF text that can be inserted into the document such as in the following fragment:

```
<CODE BEGINS>
[source,abnf]
----
{`show rulename`}
----
<CODE ENDS>
```

Figure 9

is rendered as

```
rulename = %d97 %d98 %d99
```

Figure 10

See Appendix B.2.1 for access to the source of the retrofitted specification for [RFC5234].

3.3.2. Augmented Packet Header Diagrams (APHD)

Augmented Packet Header Diagram [I-D.mcquistin-augmented-ascii-diagrams] (APHD) is a formal language used to describe a binary data layout in a machine-readable format.

The [I-D.mcquistin-augmented-ascii-diagrams] document will be retrofitted as a computerate specification to provide an internal Domain Specific Language (DSL) that permits specifying an APHD for a specification. The partial encoding of an example from section 4.1 looks like this:

```
<CODE BEGINS>
> ipv4 : Aphd
> ipv4 = MkAphd "IPv4 Header" [
>   MkField "Version" (Just "V") (Number 4) Bits "This is a" ++
>     " fixed-width field, whose full label is shown in the" ++
>     " diagram.  The field's width -- 4 bits -- is given in" ++
>     " the label of the description list, separated from the" ++
>     " field's label by a colon.",
> ...
>   MkField "Options" Nothing (Mult (Sub (Name "IHL") (Number 5))
>     (Number 32)) Bits "This is a variable-length field, whose" ++
>     " length is defined by the value of the field with short" ++
>     " label IHL (Internet Header Length).  Constraint" ++
>     " expressions can be used in place of constant values: the" ++
>     " grammar for the expression language is defined in" ++
>     " Appendix A.1.  Constraints can include a previously" ++
>     " defined field's short or full label, where one has been" ++
>     " defined.  Short variable-length fields are indicated by" ++
>     " \"...\" instead of a pipe at the end of the row."
> ...
> ]
<CODE ENDS>
```

Figure 11

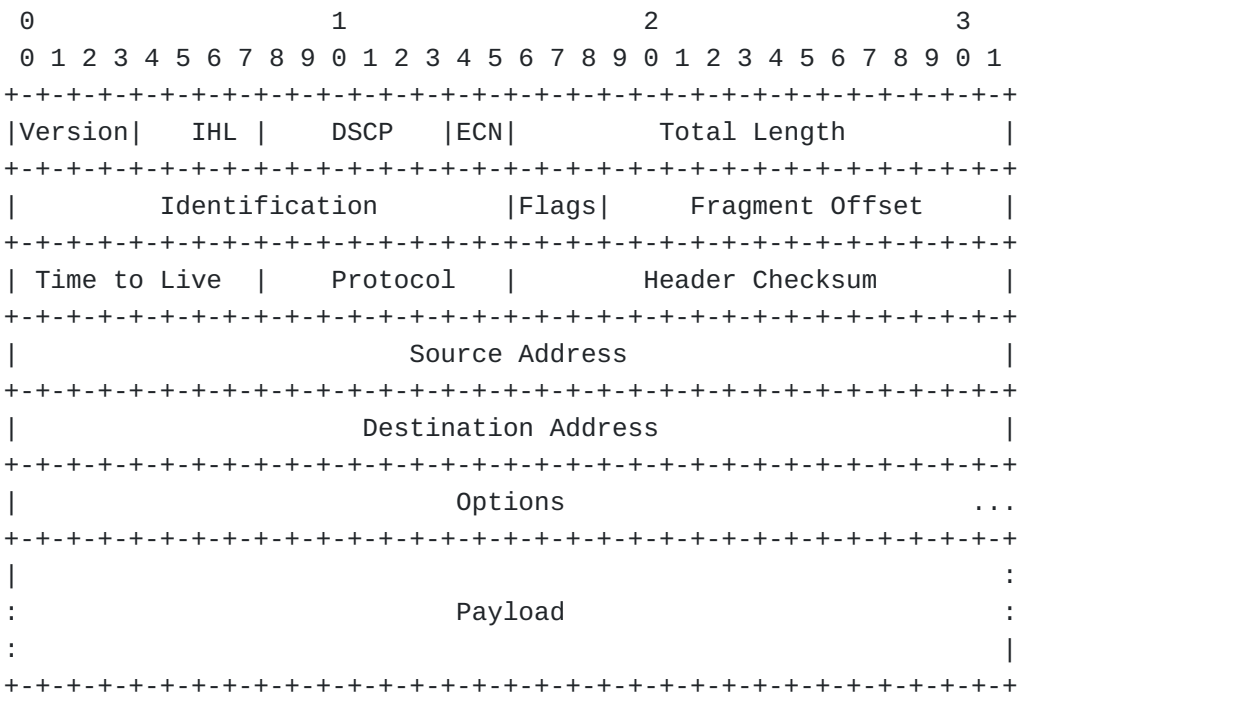
A serializer, also defined in the same specification, permits converting that description into proper ABNF text that can be inserted into the document such as in the following fragment:

```
<CODE BEGINS>
 ....
 {`show ipv4`}
 ....
<CODE ENDS>
```

Figure 12

is rendered as

An IPv4 Header is formatted as follows:



where:

Version (V): 4 bits. This is a fixed-width field, whose full label is shown in the diagram. The field's width -- 4 bits -- is given in the label of the description list, separated from the field's label by a colon.

...

Options: (IHL-5)*32 bits. This is a variable-length field, whose length is defined by the value of the field with short label IHL (Internet Header Length). Constraint expressions can be used in place of constant values: the grammar for the expression language is defined in Appendix A.1. Constraints can include a previously defined field's short or full label, where one has been defined. Short variable-length fields are indicated by "..." instead of a pipe at the end of the row.

...

Figure 13

Here the serializer generates an instance of the intermediary AsciiDoc type that describes the header line (which can be concatenated to previous lines), the block containing the diagram itself, and then a list of all the field definitions.

3.3.3. Mathematical Formulas

AsciiDoc supports writing equations using either asciimath or latexmath. The rendering for RFCs will generate an artwork element that contains both the text version of the equation and a graphical version in an SVG file.

NOTE: Not sure what to do with inline formulas, as we cannot generate an artwork element in that case.

An Idris type will be used to described equations at the type level. An interpreter will be used to calculate and insert examples in the document.

A serializer will be used to generate the asciimath code that is inserted inside a stem block.

3.3.4. RELAX NG Format

TBD

3.3.5. ASN.1

TBD

3.3.6. TLS Description Language

TBD

3.4. Proofs for Syntax

The kind of proofs that one would want in a specification are related to isomorphism, i.e. a guarantee that two or more descriptions of a data layout contain exactly the same information.

3.4.1. Isomorphism Between Type and Formal Language

We saw above that when a data layout is described with a formal language, we end up with two descriptions of that data layout, one using the Idris dependent type (and used to generate examples) and one using the formal language.

Proving isomorphism requires generating an Idris type from the formal language instance, which is done using an Idris elaborator script.

In Idris, Elaborator Reflection [Elab] is a metaprogramming facility that permits writing code generating type declarations and code (including proofs) automatically.

For instance the ABNF language is itself defined using ABNF, so after converting that ABNF into an instance of the Syntax type (which is an holder for a list of instances of the Rule type), it is possible to generate a suite of types that represents the same language:

```
<CODE BEGINS>
> abnf : Syntax
> abnf = MkSyntax [
>   "rulelist" `Eq` (Repeat (Just 1) Nothing (Group (Altern
>     (TermName "rule") (Group (Concat (Repeat Nothing Nothing
>     (TermName "c-wsp")) (TermName "c-nl") [])) [])))),
>     ...
>   ]
>
> %runElab (generateType "Abnf" abnf)
<CODE ENDS>
```

Figure 14

The result of the elaboration can then be used to construct a value of type Iso, which requires four total functions, two for the conversion between types, and another two to prove that sequencing the conversions results in the same original value.

The following example generates an Idris type "SessionDescription" from the SDP ABNF. It then proves that this type and the Sdp type contain exactly the same information (the proofs themselves have been removed, leaving only the propositions):

```
<CODE BEGINS>
> import Data.Control.Isomorphism
>
> sdp : Syntax
> sdp = MkSyntax [
>   "session-description" `Eq` (Concat (TermName "version-field")
>     (TermName "origin-field") [
>       TermName "session-name-field",
>       Optional (TermName "information-field"),
>       Optional (TermName "uri-field"),
>       Repeat Nothing Nothing (TermName "email-field"),
>       Repeat Nothing Nothing (TermName "phone-field"),
>       Optional (TermName "connection-field"),
>       Repeat Nothing Nothing (TermName "bandwidth-field"),
>       Repeat (Just 1) Nothing (TermName "time-description"),
>       Optional (TermName "key-field"),
>       Repeat Nothing Nothing (TermName "attribute-field"),
>       Repeat Nothing Nothing (TermName "media-description")
>       ]),
>   ...
>   ]
>
> %runElab (generateType "Sdp" sdp)
>
> same : Iso Sdp SessionDescription
> same = MkIso to from toFrom fromTo
>   where
>     to : Sdp -> SessionDescription
>
>     from : SessionDescription -> Abnf
>
>     toFrom : (x : SessionDescription ) -> to (from x) = x
>
>     fromTo : (x : Sdp) -> from (to x) = x
>
<CODE ENDS>
```

Figure 15

As stated in Section 3.3, the Idris type and the type generated from the formal language are not always isomorphic, because some constraints cannot be expressed in that formal language. In that case isomorphism can be used to precisely define what is missing information in the formal language type. To do so, the generated type is augmented with a delta type, like so:

```
<CODE BEGINS>
> data DeltaSessionDescription : Type where
>   ...
>
> same : Iso Sdp (SessionDescription, DeltaSessionDescription)
>   ...
<CODE ENDS>
```

Figure 16

Then the DeltaSessionDescription type can be modified to include the missing information until the same function type checks. After this we have a guarantee that we know all about the constraints that cannot be encoded in that formal language, and can check manually that each of them is described as comment.

Idris elaborator scripts will be developped for each formal languages.

3.4.2. Data Format Conversion

For specifications that describe a conversion between different data layouts, having a proof that guarantees that no information is lost in the process can be beneficial. For instance, we observe that syntax encoding tends to be replaced each ten years or so by something "better". Here again isomorphism can tell us exactly what kind of information we lost and gained during that replacement.

Here, for example, the definition of a function that would verify an isomorphism between an XML format and a JSON format:

```
<CODE BEGINS>
> isXmlAndJsonSame: Iso (XML, DeltaXML) (JSON, DeltaJson)
>   ...
<CODE ENDS>
```

Figure 17

Here DeltaXML expresses what is gained by switching from XML to JSON, and DeltaJson expresses what is lost.

3.4.3. Interoperability with Previous Versions

The syntax of the data layout may be modified as part of the evolution of a standard. In most case a version number prevents old format to be used with the new format, but in cases where that it is not possible, the new specification can ensure that both formats can co-exist by using the same techniques as above.

Conversely these techniques can be used during the design phase of a new version of a format, to check if a new version number is warranted.

3.4.4. Postel's Law

> Be conservative in what you do, be liberal in what you accept from others.
> — *Jon Postel*

One of the downsides of formal specifications is that there is no wiggle room possible when implementing it. An implementation either conforms to the specification or does not.

One analogy would be specifying a pair of gears. If one decides to have both of them made with tolerances that are too small, then it is very likely that they will not be able to move when put together. A bit of slack is needed to get the gear smoothly working together but more importantly the cost of making these gears is directly proportional to their tolerance. There is an inflexion point where the cost of an high precision gear outweighs its purpose.

We have a similar issue when implementing a formal specification, where having an absolutely conformant implementation may cost more money than it is worth spending. On the other hand a specification exists for the purpose of interoperability, so we need some guidelines on what to ignore in a formal specification to make it cost effective.

Postel's law proposes an informal way of defining that wiggle room by actually having two different specifications, one that defines a data layout for the purpose of sending it, and another one that defines a data layout for the purpose of receiving that data layout.

Existing specifications express that dichotomy in the form of the usage of SHOULD/SHOULD NOT/RECOMMENDED/NOT RECOMMENDED [RFC2119] keywords. For example the SDP spec says that "[t]he sequence CRLF (0x0d0a) is used to end a line, although parsers SHOULD be tolerant and also accept lines terminated with a single newline character." This directly infers two specifications, one used to define an SDP when sending it, that enforces using only CRLF, and a second specification, used to define an SDP when receiving it (or parsing it), that accepts both CRLF and LF.

Note that the converse is not necessarily true, i.e. not all usages of these keywords are related to Postel's Law.

To ensure that the differences between the sending specification and the receiving specification do not create interoperability problems,

we can use a variant of isomorphism, as shown in the following example (data constructors and code elided):

```
<CODE BEGINS>
> data Sending : Type where
>
> data Receiving : Type where
>
> to : Sending -> List Receiving
>
> from : Receiving -> Sending
>
> toFrom : (y : Receiving) -> Elem y (to (from y))
>
> fromTo : (y : Sending) -> True = all (== y) [from x | x <- to y]
>
<CODE ENDS>
```

Figure 18

Here we define two data types, one that describes the data layout that is permitted to be sent (Sending) and one that describes the data layout that is permitted to be received (Receiving). For each data layout that is possible to send, there is one or more matching receiving data layouts. This is expressed by the function to that takes as input one Sending value and returns a list of Receiving values.

Conversely, the from function maps a Receiving data layout onto a Sending data layout. Note the asymmetry there, which prevents to use a standard proof of isomorphism.

Then the toFrom and fromTo proofs verify that there is no interoperability issue by guaranteeing that each Receiving value maps to one and only one Sending instance and that this mapping is isomorphic.

All of this will provide a clear guidance of when and where to use a SHOULD keyword or its variants, without loss of interoperability.

As an trivial example, the following proves that accepting LF characters in addition to CRLF characters as end of line markers does not break interoperability:

```
<CODE BEGINS>
> data Sending : Type where
>   S_CRLF : Sending
>
> Eq Sending where
>   (==) S_CRLF S_CRLF = True
>
> data Receiving : Type where
>   R_CRLF : Receiving
>   R_LF : Receiving
>
> to : Sending -> List Receiving
> to S_CRLF = [R_CRLF, R_LF]
>
> from : Receiving -> Sending
> from R_CRLF = S_CRLF
> from R_LF = S_CRLF
>
> toFrom : (y : Receiving) -> Elem y (to (from y))
> toFrom R_CRLF = Here
> toFrom R_LF = There Here
>
> fromTo : (y : Sending) -> True = all (== y) [from x | x <- to y]
> fromTo S_CRLF = Refl
<CODE ENDS>
```

Figure 19

3.5. Extended Registries

Often parts of a data layout are left unspecified, so they can be defined in separate specifications. This is mainly used for extensibility purpose.

In most cases, an external registry is maintained with the list of specifications that can be used to make sense of these unspecified parts.

To make sense of an unspecified part in a data layout, 3 pieces of information are needed:

- The specification that defined the registry.
- The content of the registry itself.
- Each specification that defined one element of that registry.

Defining a data layout for a protocol means also bringing together these pieces of information so a Sum type of all the specified parts can be put together. Unfortunately the information available in

these registry is not sufficient to to bring the exact data layout from the specification of a single element in that Sum type. One reason if that multiple data layouts can be defined in the computerate specification of a standard, but there is no indication in the registry to pin point the exact data type.

For instance IANA is the organization that is maintaining the registries for the IETF. The Assigned Internet Protocol Number (https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml) is an example of a registry that contains the list of all protocols that can be carried by the Internet Protocol, and was most recently defined by RFC 5237. The first entry in that registry was defined by RFC 8200, the next on by RFC 791, and so on for each entry of that registry. RFC 8200 contains the description for multiple entries in the registry, and so an additional mechanism is needed to differentiate them.

That additional mechanism is abstracted as an extended registry that complements the existing registry, but for the sole purpose of generating that Sum type. This abstract registry is filled by information coming from the type-checking of the data layout types in the respective computerate specifications. It links a specific entry in the registry with the type pf the data layout in the specification.

When the specification for a registry is type-checked, it automatically download the external registry, and access the abstract extended registry. For each extended entry, the type-checker can automatically generate a data constructor that reference the data layout defining in the entry specification. For entries that does not have an extension (i.e. no computerate specifications exist at this time), a placeholder data constructor is generated.

Appendix C describes an implementation of the extended registry mechanism.

4. Semantics

The semantics of a communication protocol determine what messages are exchanged over a communication link and the relationship between them. The semantics are generally described only in the context of the layer that this particular protocol is operating at.

4.1. Typed Petri Nets

The semantics of a specification require to define an Idris type that strictly enforces these semantics. This can be done in an ad hoc way [Type-Driven], particularly by using linear types that express resources' consumption.

But a better solution is to design these graphically, particularly by using Petri Nets. This specification defines a DSL that permits describing a Typed Petri Net (TPN) which is heavily influenced by Coloured Petri Nets [CPN] (CPN). A CPN adds some restriction on the types that can be used in a Petri Net because of limitations in the underlying programming language, SML. The underlying programming used in TPN, Idris, does not have these limitations, so any well-formed Idris type (including polymorphic, linear and dependent types) can be directly used in TPN.

NOTE 1: A graphical editor for TPN is planned as part of the integration tooling. The graphical tool will use the document directly as storage.

Here's an example of TPN (from figure 2.10 in [CPN]):

```
<CODE BEGINS>
> NO : Type
> NO = Int
>
> DATA : Type
> DATA = String
>
> NOxDATA : Type
> NOxDATA = (NO, DATA)
>
> PTS : Place
> PTS = MkPlace "Packets To Send" NOxDATA (\() => [(1, "COL"),
>   (2, "OUR"), (3, "ED "), (4, "PET"), (5, "RI "), (6, "NET")])
>
> NS : Place
> NS = MkPlace "NextSend" NO (\() => [1])
>
> A : Place
> A = MkPlace "A" NOxDATA (\() => [])
>
> input1 : Input
> input1 = MkInput PTS (NO, DATA) pure
>
> input2 : Input
> input2 = MkInput NS NO pure
>
> output1 : Output
> output1 = MkOutput PTS (NO, DATA) pure
>
> output2 : Output
> output2 = MkOutput NS NO pure
>
> output3 : Output
> output3 = MkOutput A (NO, DATA) pure
>
> sendPacket : Transition
> sendPacket = MkTransition [input1, input2] [output1, output2,
>   output3] (\((n, d), n') => if n == n'
>                              then pure ((n, d), n, (n, d))
>                              else empty)
<CODE ENDS>
```

Figure 20

NOTE: The DSL is being currently designed, so the example shows the generated value.

From there it is easy to generate (using the non-deterministic monad in Idris) an interpreter for debugging and simulation purposes:

```
<CODE BEGINS>
> interpret : MS NOxDATA -> MS NO -> MS NOxDATA ->
>   ND (MS NOxDATA, MS NO, MS NOxDATA)
> interpret pts ns a = do
>   (pts1, pts2) <- sel pts
>   (ns1, ns2) <- sel ns
>   i1 <- input' input1 pts1
>   i2 <- input' input2 ns1
>   (pts3, ns3, a3) <- transition' sendPacket (i1, i2)
>   let o1 = output' output1 pts3
>   let o2 = output' output2 ns3
>   let o3 = output' output3 a3
>   pure (o1 ++ pts2, o2 ++ ns2, o3 ++ a)
<CODE ENDS>
```

Figure 21

NOTE: Replace by the generic variant of the interpreter.

A Petri Net has the advantage that the same graph can be reused to derive other Petri Nets, e.g., Timed Petri Nets (that can be used to collect performance metrics) or Stochastic Petri Nets.

NOTE 2: The traditional way of verifying a Petri Net is by using model checking. There is nothing in the design that prevents doing that, but because that takes quite some time to run and so cannot be part of the document processing, how do we store in the document a proof that the model checking was successful?

A TPN that covers a whole protocol (i.e. client, network, and server) is useful to prove the properties listed in the previous sections. But the TPN is also designed in a way that each of these parts can be defined separately from the others (making it a Hierarchical TPN).

4.2. Semantics Examples

Semantics examples can be wrong, so it is useful to be sure that they match the specification.

4.2.1. Data Type

As explained above, semantics can be described in an ad hoc manner, or using the TPN DSL.

4.2.2. Serializer

At the difference of syntax, where there are more or less as many ways to display them than there are syntaxes, semantics examples generally use sequence diagrams, eventually augmented with the

content of the packets exchanged (and so using the techniques described in Section 3.1).

Similarly to what is done in Section 3.3.2, an Asciidoctor block processor similar to the "msc" type of diagram used by the asciidoctor-diagram extension will be designed.

NOTE 1: We unfortunately cannot reuse the asciidoctor-diagram extension because it cannot generate both text and SVG versions of a sequence diagram.

The serializer for an example derived from a TPN generates the content of the msc AsciiDoc block, by selecting one particular path and its associated bindings through the Petri Net.

NOTE 2: We probably want to use AsciiDoc callouts for these, although that would require a modification in AsciiRfc. In fact callout would be a far better technique for other diagrams, like AAD, as it will let the renderer take care of the best way to place elements depending on the output format.

4.2.3. Presentation Format

TBD.

4.3. Formal Semantics Language

Some specifications use a formal language to describe the state machines. One shared property of these languages is that they cannot always formalize all the constraints of specific semantics, so they have to be enriched with comments. One consequence of this is that they cannot be used as a replacement for the Idris data type described in Section 4.1, a data type that is purposely complete.

4.3.1. Cosmogol

Cosmogol [I-D.bortzmeyer-language-state-machines] is a formal language designed to define states machines. The Internet-Draft will be retrofitted as a computerate specification to provide an internal Domain Specific Language (DSL) that permits specifying an instance of that language. A serializer and elaborator script will also be defined.

Finally, an Asciidoctor block processor would be used to convert the language into both a text and a graphical view of the state machine.

NOTE: Add examples there.

4.4. Proofs for Semantics

Like for syntax formal languages, an elaborator script permits generating a type from a TPN instance. That type can then be used to write proofs of the properties that we expect from the semantics.

4.4.1. Isomorphism

An isomorphism proof can be used between two types derived from the semantics of a specification, for example to prove that no information is lost in the converting between the underlying processes, or when upgrading a process.

An example of that would be to prove (or more likely disprove) that the SIP state machines are isomorphic to the WebRTC state machines.

4.4.2. Postel's Law

Like for the syntax, semantics can introduce wiggle room between the state machines on the sending side and the state machines on the receiving side. A similar isomorphism proof can be used to ensure that this is done without loss of interoperability.

4.4.3. Termination

The TPN type can be used to verify that the protocol actually terminates, or that it always returns to its initial state. This is equivalent to proving that a program terminates.

4.4.4. Liveness

The TPN type can be used to verify that the protocol is productive, i.e. that it does not loop without making progress.

5. Verified Code

When applied, the techniques described in Section 3 and Section 4 result in a formal specification, in the form of a set of types. Types are logical propositions so proofs and disproofs can be written about them. Interpreted as code, these eventual proofs happen to be proofs that a specification is implementable.

To make these pieces of code composable, a specification is split in multiple modules, each one represented as a unique function. The type of each of these functions is derived from the hierarchical TPNs described in Section 4, by bundling together all the inputs of the TPN module as the input for that function, and bundling all the outputs of the TPN module as the output of this function.

For instance the IPv4 layer is really 4 different functions:

- A function that converts between a byte array and a tree representation (parsing).
- A function that takes a tree representation and a maximum MTU and returns a list of tree representations, each one fitting inside the MTU.
- A function that accumulates tree representations of an IP fragment until a tree representation of a full IP packet can be returned.
- A function that convert a tree representation into a byte array.

The description of each function is incomplete, as in addition to the input and the output listed, these functions needs some ancillary data, in the form of:

- state, which is basically values stored between evaluations of a function,
- an optional signal, that can be used as an API request or response. As timers are a fundamental building block for communication protocols, one common uses for that signal are to request the arming of a timer, and to receive the indication of the expiration of that timer.

To unclutter the signature of these function, these ancillary data are passed to the function using an ad-hoc monad named the Impl Monad (Impl is short for Implementation or Implementable). To unclutter even further the signature of these functions, that same Impl monad can be used to store the various proofs consumed and produced by these functions, making these function more easily composable.

6. Bibliography

[RFC0761] Postel, J., "DoD standard Transmission Control Protocol", IETF RFC 0761, DOI 10.17487/RFC0761, January 1980, <https://www.rfc-editor.org/info/rfc761>.

[RFC7991] Hoffman, P., "The "xml2rfc" Version 3 Vocabulary", IETF RFC 7991, DOI 10.17487/RFC7991, December 2016, <https://www.rfc-editor.org/info/rfc7991>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", IETF RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC1034]
Mockapetris, P., "Domain names - concepts and facilities", IETF RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC8489] Petit-Huguenin, M., Salgueiro, G., Rosenberg, J., Wing, D., Mahy, R., and P. Matthews, "Session Traversal Utilities for NAT (STUN)", IETF RFC 8489, DOI 10.17487/RFC8489, February 2020, <https://www.rfc-editor.org/info/rfc8489>.

[RFC8656] Reddy, T., Ed., Johnston, A., Ed., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", IETF RFC 8656, DOI 10.17487/RFC8656, February 2020, <https://www.rfc-editor.org/info/rfc8656>.

[I-D.bortzmeyer-language-state-machines]
Bortzmeyer, S., "Cosmogol: a language to describe finite state machines", IETF I-D.bortzmeyer-language-state-machines, November 2006.

[I-D.mcquistin-augmented-ascii-diagrams]
McQuistin, S., Band, V., and C. Perkins, "Describing Protocol Data Units with Augmented Packet Header Diagrams", IETF I-D.mcquistin-augmented-ascii-diagrams, February 2020.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", IETF RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[I-D.ribose-asciirfc] Tse, R., Nicholas, N., and P. Brasolin, "AsciiRFC: Authoring Internet-Drafts And RFCs Using AsciiDoc", IETF I-D.ribose-asciirfc, April 2018.

[Curry-Howard] "Curry-Howard correspondence", <https://en.wikipedia.org/wiki/Curry-Howard_correspondence>.

[Zave] "Experiences with Protocol Description", October 2011.

[Knuth92] "Literate Programming", 1992.

[Elab] "Elaborator reflection: extending Idris in Idris", 2016.

[AsciiDoc] "AsciiDoc", <https://en.wikipedia.org/wiki/AsciiDoc>.

[Asciidoctor] "Asciidoctor", <https://asciidoctor.org/docs/user-manual/>.

[Metanorma]
"Metanorma", <https://www.metanorma.com/>.

[Idris] "Idris: A Language with Dependent Types", <https://www.idris-lang.org/>.

[Literate] "Literate programming", <http://docs.idris-lang.org/en/latest/tutorial/miscellany.html#literate-programming>.

[Blockquotes] "Markdown-style blockquotes", <https://asciidoctor.org/docs/user-manual/#markdown-style-blockquotes>.

[CPN] "Coloured Petri Nets: Modelling and Validation of Concurrent Systems", 2009.

[Type-Driven] "Type-Driven Development with Idris", 2017.

[Brinkmann02] "RTL-datapath verification using integer linear programming", 2009.

Appendix A. Command Line Tools

A.1. Installation

The computerate command line tools are run inside a Docker image, so the first step is to install the Docker software or verify that it is up to date (https://docs.docker.com/install/).

Note that for the usage described in this document there is no need for Docker EE or for having a Docker account.

The following instructions assume a Unix based OS, i.e. Linux or MacOS. Lines separated by a "\" character are meant to be executed as one single line, with the "\" character removed.

A.1.1. Download the Docker Image

To install the computerate tools, the fastest way is to download and install the Docker image using a temporary image containing the dat tool:

```
docker pull veggiemonk/dat-docker
mkdir computerate
cd computerate
docker run --rm -u $(id -u):$(id -g) -v \
 $(pwd):/tools veggiemonk/dat-docker dat clone \
 dat://6a33cb289d5818e3709f62e95df41be537edba5f4dc26593e2cb870c7982345b
 tools
```

Figure 22

After this, the image can be loaded in Docker. The newly installed Docker image also contains the dat command, so there is no need to keep the veggiemonk/dat-docker image after this:

```
docker load -i tools.tar.xz
docker image rm --force veggiemonk/dat-docker
```

Figure 23

A new version of the tools is released at the same time a new version of this document is released. After this, running the following command in the computerate directory will pull any new version of the tool tar file:

```
docker run --rm -u $(id -u):$(id -g) \
 -v $(pwd):/computerate computerate/tools dat pull --exit
```

Figure 24

The docker image can then be loaded as above.

A.2. Using the computerate Command

The Docker image main command is computerate, which takes the same parameters as the metanorma command from the Metanorma tooling:

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools computerate -t ietf -x txt <file>
```

Figure 25

The differences with the metanorma command are:

* The computerate command can process Literate Idris files (files with a "lidr" extension, aka lidr files), in addition to AsciiDoc files (files with an "adoc" extension, aka adoc files). When a lidr file is processed, all embedded code fragments (text between prefix "{`" and suffix "`}") are evaluated in the context of the Idris code contained in this file. Each code fragment (including the prefix and suffix) are then substituted by the result of that evaluation.

* The computerate command can process included lidr files in the same way. The embedded code fragments in the imported file are processed in the context of the included lidr file, not in the context of the including file. Idris modules (either from an idr or lidr file) can be imported the usual way.

*The literate code (which is all the text that is starting by a ">" symbol in column 1) in a lidr file will not be part of the rendered document.

*The computerate command can process transclusions, as explained in Section 2.2.

*Lookup of external references is disabled. Use either raw XML references or an external directory.

*Instead of generating a file based on the name of the input file, the computerate command generates a file based on the :name: attribute in the header of the document.

The computerate command can also be used to generate an xmlrfc v3 file, ready for submission to the IETF:

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools computerate -t ietf -x rfc <file>
```

Figure 26

A.3. Using the Idris REPL

idr and lidr files can be loaded directly in the Idris REPL for debugging:

```
docker run --rm -it -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools idris <lidr-file>
```

Figure 27

It is possible to directly modify the source code in the REPL by entering the :e command, which will load the file in an instance of VIM preconfigured to interact with the REPL.

Alternatively the Idris REPL can be started as a server:

```
docker run --rm -it -u $(id -u):$(id -g) -p 127.0.0.1:4294:4294 \
  -v $(pwd):/computerate computerate/tools idris
```

Figure 28

Then if a source file is loaded in a separate console with the VIM instance inside the Docker image, it can interact with the REPL server:

```
docker run --rm -u $(id -u):$(id -g) --net=host \
  -v $(pwd):/computerate computerate/tools vim <file>
```

Figure 29

Note that both commands must be run from the same directory.

A.4. Using Other Commands

For convenience, the docker image provides the latest version of the xml2rfc, aspell, and idnits tools.

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools xml2rfc
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools idnits --help
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools aspell
```

Figure 30

The Docker image also contains a extended version of git that will be used to retrieve the computerate specifications in Appendix B.

A.5. Bugs and Workarounds

*Errors in embedded code do not stop the process but replace the text by the error message, which can be easily overlooked.

*backticks are not escaped in code fragments.

*The current version of Docker in Ubuntu fails, but this can be fixed with the following commands:

```
sudo apt-get install containerd.io=1.2.6-3
sudo systemctl restart docker.service
```

Figure 31

*The Asciidoctor processor does not correctly format the output in all cases (e.g. "++"). The escaping can be done in Idris until this is fixed.

*Sometimes the Idris processing fails with an error "Module needs reloading". Deleting all the files with the ibc extension will solve that problem.

*Trying to fetch nonexistent new commits on a git repository will block for 12 seconds.

A.6. TODO List

*Embedded blocks.

*Test on Windows.

*Using recursive modules with Idris.

Appendix B. Computerate Specifications Library

B.1. Installation

The git repositories that compose the Computerate Specification Library are distributed over a peer-to-peer protocol based on dat.

This requires an extension to git, extension that is already installed in the Docker image described in Appendix A. The following command can be used to retrieve a computerate specification:

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools git clone --recursive dat://<public-key> <name>
```

Figure 32

Here <public-key> is the dat public key for a specific computerate specification and <name> is the recommended name. Do not use the dat URIs given in Appendix A, as only the dat public keys listed in Appendix B.2 can be used with a git clone.

Updating the repository also requires using the Docker image:

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools git pull
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools git submodule update
```

Figure 33

All the git commands that do not require access to the remote can be run natively or from the Docker image.

Note that for the computerate specification library the computerate command must be run from the directory that is one level above the git repository. The name of the root document is always Main.adoc, or rarely Main.lidr:

```
docker run --rm -u $(id -u):$(id -g) -v $(pwd):/computerate \
  computerate/tools computerate -t ietf -x txt \
  <git-repository>/Main.lidr
```

Figure 34

B.2. Catalog

For the time being this document will serve as a catalog of available computerate specifications.

B.2.1. RFC 5234

```
Name: RFC5234
Public key:
  994e52b29a7bf4f7590b0f0369a7d55d29fb22befd065e462b2185a8207e21f1
```

Figure 35

Appendix C. Extended Registry

When a data layout is typechecked, a Type Provider is used to write the data layout in the extended registry.

```
%provide (T : Type) with extendRegistry
 "https://www.iana.org/assignments/cose/cose.xhtml#algorithms"
 "1" Icmp
```

Figure 36

The extendRegistry function takes 3 parameters:

- *The URL or identifier for the registry to extend.
- *A unique identifier for the entry to extend. The entry must exist.
- *The data layout Type to bind with that entry.

In the computerate specification that defines a registry, the following code will automatically generates the corresponding Sum type:

```
%provide (MyType : Type) with registry
 "https://www.iana.org/assignments/cose/cose.xhtml#algorithms"
```

Figure 37

That code will download the current content of the registry identified by the URL passed as parameter, will locate the extended registry, if any, and will generate the Sum type of type MyType.

The implementations of the extendRegistry and registry functions currently only recognize IANA registries.

Appendix D. Errata Statistics

In an effort to quantify the potential benefits of using formal methods at the IETF, an effort to relabel the Errata database is under way.

The relabeling uses the following labels:

Label	Description
AAD	Error in an ASCII bit diagram
ABNF	Error in an ABNF
Absent	The errata was probably removed
ASN.1	Error in ASN.1
C	Error in C code
Diagram	Error in a generic diagram
Example	An example does not match the normative text
Formula	Error preventable by using Idris code
Ladder	Error in a ladder diagram
Rejected	The erratum was rejected
Text	Error in the text itself, no remedy
TLS	Error in the TLS language

Table 1

At the time of publication the first 700 errata, which represents 11.88% of the total, have been relabeled. On these, 34 were rejected and 27 were deleted, leaving 639 valid errata.

Label	Count	Percentage
Text	396	61.97%
Formula	66	10.32%
Example	64	10.0%
ABNF	38	5.94%
AAD	32	5.00%
ASN.1	27	4.22%
C	9	1.40%
Diagram	4	0.62%
TLS	2	0.31%
Ladder	1	0.15%

Table 2

Note that as the relabeling is done in in order of erratum number, at this point it covers mostly older RFCs. A change in tooling (e.g. ABNF verifiers) means that these numbers may drastically change as more errata are relabeled. But at this point it seems that 38.02% of errata could have been prevented with a more pervasive use of formal methods.

Acknowledgements

Thanks to Jim Kleck, Stephane Bryant, Eric Petit-Huguenin, Nicolas Gironi, Stephen McQuistin, and Greg Skinner for the comments, suggestions, questions, and testing that helped improve this document.

Changelog

*draft-petithuguenin-computerate-specifying-03

-Document

oNotes are now correctly displayed.

oAdd "Implementations Oriented Standards" section.

oAdd "Extended Registries" section and appendix.

oAdd paragraph about hierarchical petri nets.

oConvert "Verified Code" section into a top level section, and expand it.

oAdd "Implementation-Oriented Standards" section.

-Tooling

oMany bug fixes in metanorma-ietf.

oUpdate xml2rfc to 2.40.1.

oRebuilding text for an RFC with xml2rfc now uses pagination.

oUpdate metanorma-ietf to version 2.0.5.

oThe "computerate" command can directly generate PDF files.

oAdd support in xml2rfc for generating PDF files.

oAdd asciidoctor-revealjs.

oUpdate metanorma to version 1.0.0.

oUpdate metanorma-cli to version 1.2.10.1.

-Library

oNo update

*draft-petithuguenin-computerate-specifying-02

-Document

oSwitch to rfcxml3.

oStatus is now experimental.

oMany nits.

oFix incorrect errata stats.

oMove acknowledgment section at the end.

oRewrite the APHD section (formerly known as AAD) to match draft-mcquistin-augmented-diagrams-01.

oFix non-ascii characters in the references.

oIntermediate AsciiDoc representation for serializers.

-Tooling

oxmlrfc3 is now the default extension.

o"docName" and "category" attributes are now generated, and the "prepTime" is removed.

oUpdate xml2rfc to 2.35.0.

oRemove LanguageTool.

oUpdate Metanorma to version 0.3.17.

oUpdate Asciidoctor to 2.0.10.

oUpdate list of Working Groups.

-Library

oNo update.

*draft-petithuguenin-computerate-specifying-01

-Document

oNew changelog appendix.

oFix incorrect reference, formatting in Idris code.

oAdd option to remove container in all docker run command.

- oAdd explanations to use the Idris REPL and VIM inside the Docker image.

- oAdd placeholders for ASN.1 and RELAX NG languages.

- oNew Errata appendix.

- oNits.

- oImprove Syntax Examples section.

-Tooling

- oUpdate Metanorma to version 0.3.16

- oUpdate MetaNorma-cli to version 1.2.7.1

- oSwitch to patched version of Idris 1.3.2 that supports remote REPL in Docker.

- oAdd VIM and idris-vim extension.

- oRemove some debug statements.

-Library

- oNo update

Author's Address

Marc Petit-Huguenin
Impedance Mismatch LLC

Email: marc@petit-huguenin.org