



# Formalizing a Web Standard's requirements: RSS v2.0

Rules: Logic and Applications (Dec. 19, 2018)

Konstantinos Barlas

University of West Attica



# Summary

- Take an open standard's natural language specification and
- **Formally** rewrite it
- Discussion about possible benefits
- Example – RSS v2.0



# Open Standards

- ▶ **“Open” term -> many definitions!**
  - ▶ Dictionaries, National IT agencies, IDABC, WTO, Governments, OASIS, ANSI, etc., all provide different definitions.
- ▶ **Recurring themes:**
  - ▶ **Motivation**
    - ▶ Built to encourage interoperability and help popularize new technologies
  - ▶ **Development**
    - ▶ developed by an open process
    - ▶ easy for anyone to participate in
    - ▶ Open to public input
  - ▶ **Usage**
    - ▶ easily accessible for all to read and use
    - ▶ no control or tie-in by any specific group or vendor

# Ambiguity in Natural Languages

Shoes Must  
Be Worn

Dogs Must  
Be Carried

- ▶ Precision matters when it comes to protocols
- ▶ Context is often assumed.
  - ▶ a medical appliance will assume a medical background which is usually not well defined.
- ▶ Also, during the development of the standard!
  - ▶ We need to be able to carry a set of requirements during all phases of development
  - ▶ All of these need to reflect our original intentions!



How the customer explained it



How the project leader understood it



How the engineer designed it



How the programmer wrote it



How the sales executive described it



How the project was documented



What operations installed



How the customer was billed



How the helpdesk supported it



What the customer really needed



# Formal Methods




- Techniques based on mathematics, used to:
  - describe a system
  - analyze its behavior
  - assist its design by properties' verification carried out through rigorous and effective automated reasoning tools.
- Formal specifications are expressed in a language whose vocabulary, syntax and semantics are formally defined
- Different than current specification methods



# Formal Methods



- ▶ algebraic approach of formal specifications
  - ▶ specifies a system in terms of its operations and the relationships between those operations.
  - ▶ Types of data are formally specified along with operations on those data types.
  - ▶ The implementation details, such as the size of representations are quite abstract in nature.



# Algebraic Specifications of Open Standards

- An Algebraic Formal Specification of an Open Standard;
  - Design a standard using this, or
  - Complement an existing natural language specification of a standard





# Benefits



1. Less ambiguity issues:
  - ▶ Even in a very careful standard's natural language specification what someone reads does is not always what the designers had in mind.
  - ▶ Applying a degree of formalism eliminates that problem
    - ▶ makes designers ask the right questions
    - ▶ improves the level of understanding
  - ▶ The N.L. specification of the open standard can not really be eliminated
    - ▶ more natural for humans to start with a N.L. specification
    - ▶ its involvement can be minimized.
    - ▶ it can be used to begin with (at the requirements part) and then use the formal version from there on.



# Benefits

## 2. Smaller specifications

- F.S. are significantly more compact than the ones written in natural languages.
  - The specification for the format of ARPA Internet text messages is 40 pages.
  - A formal specification of that could be just a few pages.
- A well written specification of a small module can be applicable in other bigger systems as well.



# Benefits

3. Under circumstances (using an Algebraic Specification), we can:
- Verify the validity of the specification
    - Did we build the right system?
  - Validate an implementation of the specification
    - Did we build the system right?



# Test Case – RSS v2.0

- ▶ RSS - Really Simple Syndication
  - ▶ Extensible Markup Language (XML)-based document format
  - ▶ Allows users to avoid visiting all of the websites they are interested in
  - ▶ New content is automatically checked for and advertised as soon as it is available.
  - ▶ RSS feeds can be read using software called an “RSS reader”, “feed reader”, or “aggregator”. Can be web-based, desktop-based, or mobile-device-based.
- ▶ The RSS 2.0 specification describes how to create RSS documents.
- ▶ RSS is a dialect of XML.
  - ▶ All RSS files must conform to the XML 1.0 specification, as published on the World Wide Web Consortium (W3C) website.

# Sample RSS file

```
<rss version="2.0">
  <channel>
    <title>Liftoff News</title>
    <link>http://liftoff.msfc.nasa.gov/</link>
    <description>Liftoff to Space Exploration.</description>
    <language>en-us</language>
    <pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>
    <docs>http://blogs.law.harvard.edu/tech/rss</docs>
    <generator>Weblog Editor 2.0</generator>
    <managingEditor>editor@example.com</managingEditor>
    <webMaster>webmaster@example.com</webMaster>
    <item>
      <title>Star City</title>
      <link>http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp</link>
      <description>How do Americans get ready to work with Russians aboard the International Space Station? They take a crash course in culture, language and protocol at Russia's <a href="http://howe.iki.rssi.ru/GCTC/gctc_e.htm">Star City</a>.</description>
      <pubDate>Tue, 03 Jun 2003 09:39:21 GMT</pubDate>
      <guid>http://liftoff.msfc.nasa.gov/2003/06/03.html#item573</guid>
    </item>
  </channel>
</rss>
```



# CafeOBJ

- ▶ An executable, algebraic specification language.
  - ▶ Open source, provided free of charge under GNU GPL v2
  - ▶ Used for writing formal (i.e. mathematical) specifications of models for wide varieties of software and systems, and verifying properties of them.
  - ▶ Implements equational logic by rewriting and can be used as a powerful interactive theorem proving system.
  - ▶ Specifiers can write proof scores also in CafeOBJ and doing proofs by executing the proof scores.



# XML structures

- ▶ RSS is an XML file
- ▶ As every XML file, RSS has to follow some syntax rules (“well formed”) and conform to a specific document type (set of rules that define legal elements and attributes) – a DTD or a XML Schema
- ▶ XML2OBJ
  - ▶ CafeOBJ framework for describing XML structure
  - ▶ A module that gets imported into the specification providing XML support.
  - ▶ Adds methods that parse a XML tree structure from a file and can
    - ▶ Find a specific element (by tag and/or by parent) and return it (or its content)
    - ▶ Check whether an element has attributes and return both the names of those attributes and their assorted values



# RSS formalization

- ▶ Why?

- ▶ XML is not enough

- ▶ A DTD can't validate data – XML Schema can

- ▶ Neither DTD nor XML Schema can do complex operations:

- ▶ Compare values from 2 elements

- ▶ Reuse a constraint for every element of the same type

- ▶ etc..

- ▶ RSS' spec:

- ▶ Sometimes messy

- ▶ Unclear

- ▶ Verbose





# Reducing verbosity

- ▶ Most RSS elements are similar in nature:
  - ▶ Title, Link, Description, Copyright, ManagingEditor, ...
  - ▶ All share a very similar specification
  - ▶ What changes:
    - ▶ XML Name
    - ▶ Name of the operators
- ▶ Instead of writing so many similar modules
  - ▶ We write one module with this generic structure that serves as a building block
  - ▶ All similar modules just reuse that structure with some term-renaming
  - ▶ If needed, we add everything else (XML attributes, requirements, properties, constrictions)

```

mod* BUILDINGBLOCK {
  pr(XML)
  op getxmlcontent : ElemNdList -> String .
  op getattributes : ElemNdList -> String .
  op XMLTitlePrefix : -> XMLName .
  eq XMLTitlePrefix = "" .

  vars X X1 : XMLName .
  vars A A1 : AttNdList .
  var S : String .
  vars EL EL1 : ElemNdList .

  ceq getxmlcontent( < X A >[ tx(S) ] ) = S if (X = XMLTitlePrefix) and (A = noAtt) .
  ceq getxmlcontent( ( < X A >[ tx(S) ] ) @ EL1) = S if (X = XMLTitlePrefix) and (A = noAtt) .
  ceq getxmlcontent( ( < X A >[ tx(S) ] ) @ EL) = getxmlcontent(EL) if not((X = XMLTitlePrefix) and (A = noAtt)) .
  ceq getxmlcontent( < X A >[ EL ] ) = getxmlcontent(EL) if not((X == XMLTitlePrefix) and (A = noAtt)) .
  ceq getxmlcontent( < X A >[ ( < X1 A1 >[ EL ] ) ] ) = getxmlcontent( < X1 A1 >[ EL ] ) if not((X = XMLTitlePrefix)
and (A = noAtt)) . }

mod! TITLE {
  protecting(BUILDINGBLOCK * {op getxmlcontent -> gettitle, op XMLTitlePrefix -> TitleXMLPrefix})
  eq TitleXMLPrefix = "Title" .
}

mod! LINK {
  protecting(BUILDINGBLOCK * {op getxmlcontent -> getlink, op XMLTitlePrefix -> LinkXMLPrefix})
  eq LinkXMLPrefix = "Link" .
}

```

# Adding restrictions

- ▶ The Language element has a list of allowed values.

```
mod! LANGUAGE {
    protecting(BUILDINGBLOCK * {op getxmlcontent -> getlanguage, op
XMLTitlePrefix -> LanguageXMLPrefix})
    eq LanguageXMLPrefix = "Language" .

    op properlanguage? : String -> Bool
    var L : String .
    eq properlanguage?(L) =
        if
            L = "en-us"
            or L = "el-gr"
            or ...
        then true
        else false
    fi .
}
```



# Channel



- Main module that imports everything else
- *properchannel?* operator checks if a given channel is valid (and also describes the protocol's reqs):
  1. Root element is *Channel* – no attributes
  2. *Title*, *Link* & *Description* elements under channel – no attributes
  3. If there is a *Language* element -> no attributes and the content is within the list of accepted codes (e.g. “en-us”)

# Channel

## 4. *TTL*

- The maximum number of minutes to cache the data before an aggregator requests it again
- Indicates how long after the publication date can a feed stay alive - easy to specify if needed
- $PubDate + TTL \leq Now()$
- ??

## 5. *PubDate & LastBuildDate*

- If present → no attributes
- Date **is** proper

## 6. *Cloud*

- XML element with 5 required attributes
- If present → all 5 attributes must be present



# Channel



## 7. *SkipHours & SkipDays*

- ▶ Contains up to 24 *<hour>* sub elements containing a time when aggregators may not read the channel.
- ▶ *Properchannel?* Compares the current hour with those sub elements.
- ▶ SkipDays is similar

## 8. *TextInput*

- ▶ *“The textInput element defines a form to submit a text query to the feed's publisher over the Common Gateway Interface (CGI) (optional).”*
- ▶ The RSS specification actively discourages publishers from using the textInput element, calling its purpose "something of a mystery" and stating that "most aggregators ignore it." Fewer than one percent of surveyed RSS feeds included the element.
- ▶ For this reason, publishers should not expect it to be supported in most aggregators.

# Channel

## 9. *Image*

- Specifies an image that can be displayed with the channel
- Contains 3 required & 3 optional sub elements.
  - $0 \leq \text{width} \leq 144$  (*pixels*)
  - $0 \leq \text{height} \leq 400$  (*pixels*)
- *Properchannel?* uses *properimage?* operator
- Title and Link elements **should (?)** have the same value as the channel's title & link.

```
eq properwidth?(W) = if ((W >= 0) and (W <= 144))
then true
else false
fi .
```



# Channel

## 10. *Item*

- ▶ Represents a “story”
- ▶ May contain any number of items.
- ▶ All sub elements are optional, but a title or a description must be present.
  - ▶ Title and description’s specifications are not declared again
- ▶ Sub elements of item have similar properties.
- ▶ *properitem?* takes care of all requirements for the *Item* element



```

eq properchannel?(< X A > [ EL ]) =
  if ( ( X = ChannelXMLPrefix ) and ( A = noAtt )
    and ( xmlnameexists?(TitleXMLPrefix, EL) and (getxmlatt(TitleXMLPrefix, EL) = noAtt) and
(getparent(TitleXMLPrefix, < X A > [ EL ]) = ChannelXMLPrefix) )
    and ( xmlnameexists?(LinkXMLPrefix, EL) and (getparent(LinkXMLPrefix, < X A > [ EL ]) =
ChannelXMLPrefix) and (getxmlatt(LinkXMLPrefix, EL) = noAtt) )
    and ( xmlnameexists?(DescriptionXMLPrefix, EL) and (getparent(DescriptionXMLPrefix, < X A > [ EL ]) =
ChannelXMLPrefix) and (getxmlatt(DescriptionXMLPrefix, EL) = noAtt) )
    and (xmlnameexists?(LanguageXMLPrefix, EL) implies ((getxmlatt(LanguageXMLPrefix, EL) = noAtt) and
properlanguage?(getlanguage(EL))) )
    ...
    and ((xmlnameexists?(PubDateXMLPrefix, EL) and (getparent(PubDateXMLPrefix, < X A > [ EL ]) =
ChannelXMLPrefix)) implies (getxmlatt(PubDateXMLPrefix, EL) = noAtt and properdate?(getpubdate(EL))) )
    and ((xmlnameexists?(TTLXMLPrefix, EL)) implies (getxmlatt(TTLXMLPrefix, EL) = noAtt))
    and (xmlnameexists?(CloudXMLPrefix, EL) implies propercloud?(returnxmlnode(CloudXMLPrefix, EL)) )
    and (xmlnameexists?(SkipHoursXMLPrefix, EL) implies ((getxmlatt(SkipHoursXMLPrefix, EL) == noAtt) and
validhour(returnxmlnode(SkipHoursXMLPrefix, EL), hour(today))))
    and (xmlnameexists?(SkipDaysXMLPrefix, EL) implies ( (getxmlatt(SkipDaysXMLPrefix, EL) == noAtt) and
validday(returnxmlnode(SkipDaysXMLPrefix, EL), dayT(today))))
    and (xmlnameexists?(TextInputXMLPrefix, EL) implies proptextinput?(returnxmlnode(TextInputXMLPrefix,
EL)))
    and (xmlnameexists?(ItemXMLPrefix, EL) implies properitem?(returnxmlnode(ItemXMLPrefix, EL)))
    and (xmlnameexists?(ImageXMLPrefix, EL) implies properimage?(returnxmlnode(ImageXMLPrefix, EL)))
    and (xmlnameexists?(LastBuildDateXMLPrefix, EL) implies ( (getxmlatt(LastBuildDateXMLPrefix, EL) == noAtt)
and properdate?(getlastbuilddate(EL))) )
  then true
  else false
fi .

```

# Providing a sample channel

```
open CHANNEL .
  op samplechannel : -> ElemNdList .
  eq samplechannel = < "Channel" noAtt > [
    ( < "Title" noAtt > [ txt("Title goes here") ] ) @
    ( < "Link" noAtt > [ txt("URL") ] ) @
    ( < "Description" noAtt > [ txt("This is the description") ] ) @
    ( < "Category" ("Domain" @= "Syndic8") > [ txt("1765") ] ) @
    ( < "Language" noAtt > [ txt("el-gr") ] ) @
    ( < "Copyright" noAtt > [ txt("Copyright 2002, Spartanburg Herald-Journal") ] ) @
    ( < "PubDate" noAtt > [ dat(date("Thu", 2013, 4, 24, 17, 22, 0, "GMT")) ] ) @
    ( < "Cloud" (("domain" @= "rpc.sys.com") @ ("port" @= "80") @ ("path" @= "/RPC2") @ ("registerprocedure" @= "pingMe") @
("protocol" @= "soap")) > [ txt("") ] ) @
    ( < "Image" noAtt > [
      ( < "Title" noAtt > [ txt("Title of the image here") ] ) @
      ( < "Link" noAtt > [ txt("Image redirects here") ] ) @
      ( < "URL" noAtt > [ txt("URL of the image") ] ) @
      ( < "Width" noAtt > [ nat(55) ] ) @
      ( < "Height" noAtt > [ nat(400) ] ) ) @
      ( < "TTL" noAtt > [nat(60) ] ) @
      ( < "Item" noAtt > [
        ( < "Title" noAtt > [ txt("Title of the 1st item here") ] ) @
        ( < "Link" noAtt > [ txt("Link of the 1st item here") ] ) @ ) @
      ( < "SkipHours" noAtt > [
        ( < "Hour" noAtt > [ nat(11) ] ) @
        ( < "Hour" noAtt > [ nat(12) ] ) ] ) @
      ( < "SkipDays" noAtt > [
        ( < "Day" noAtt > [ txt("Sunday") ] ) @
        ( < "Day" noAtt > [ txt("Tuesday") ] ) ] ) ] .
  close
```

# Reductions

```
-- opening module CHANNEL.. done.  
%CHANNEL> %CHANNEL> %CHANNEL> _  
%CHANNEL> *  
-- reduce in %CHANNEL : (properchannel?(samplechannel)):Bool  
(true):Bool  
(0.000 sec for parse, 3909 rewrites(4.150 sec), 30349 matches)
```

- Suppose we introduce an error
  - (e.g. replace the attributes of “Cloud” with noAtt)

```
-- opening module CHANNEL.. done.  
%CHANNEL> %CHANNEL> %CHANNEL> _  
%CHANNEL> *  
-- reduce in %CHANNEL : (properchannel?(samplechannel)):Bool  
(false):Bool  
(0.000 sec for parse, 3854 rewrites(4.140 sec), 30074 matches)
```

- We can turn on more detail in the output to see exactly which condition failed to evaluate properly



# How do our claims hold up?

- ▶ The specification we've created can also work as an RSS/XML DTD as we can check a sample RSS file (converted via XML2OBJ) against the specification.
- ▶ Verbosity
  - ▶ Specification: 650 (820 lines with comments) in 44 modules
  - ▶ lots of empty lines
  - ▶ The original specification: bigger in size;
    - ▶ some elements link to external pages that provide the specifications
    - ▶ Date and Time Specifications, as they appear in RFC 822, are 40 pages big.



# How do our claims hold up?

- ▶ Clarity:

- ▶ Unclear elements (TTL, title and link sub-elements of the image element).
- ▶ We can only specify what we think that the developers originally intended, but..
- ▶ writing a formal specification makes us investigate in depth each element:
  - ▶ Asking the right questions gives a clear view of intentions and helps avoid confusions as to how is the standard supposed to work

- ▶ Requirements amalgamation:

- ▶ Each requirement for the RSS file can be seen in the *properchannel?* operator. We can easily isolate any requirements we want in that operator and see how a change affects the big picture, or not.



# Discussion



- ▶ No other formalizations of open standards available. Why?
  - ▶ RSS seems a suitable candidate. What about others?
    - ▶ Perhaps a different specification approach / formal method should be used.
    - ▶ Z, VDM, etc..
    - ▶ **Many** formal method tools.
  - ▶ Slow learning curve of F.M.
    - ▶ Changing industrial habits is not happening overnight
    - ▶ Different kind of training is required
      - ▶ Positive learning experiences while teaching formal specification concepts
    - ▶ Is it worth it?



**Thank you!**

Questions?