Configuring tDOM schema objects with context-dependent Tcl commands

Rolf Ade
Freelancer

Tcl/Tk, XML, C and
Product-Data Expert

Stuttgart, Germany
rolf@pointsman.de

This presentation title sounds complicated ...

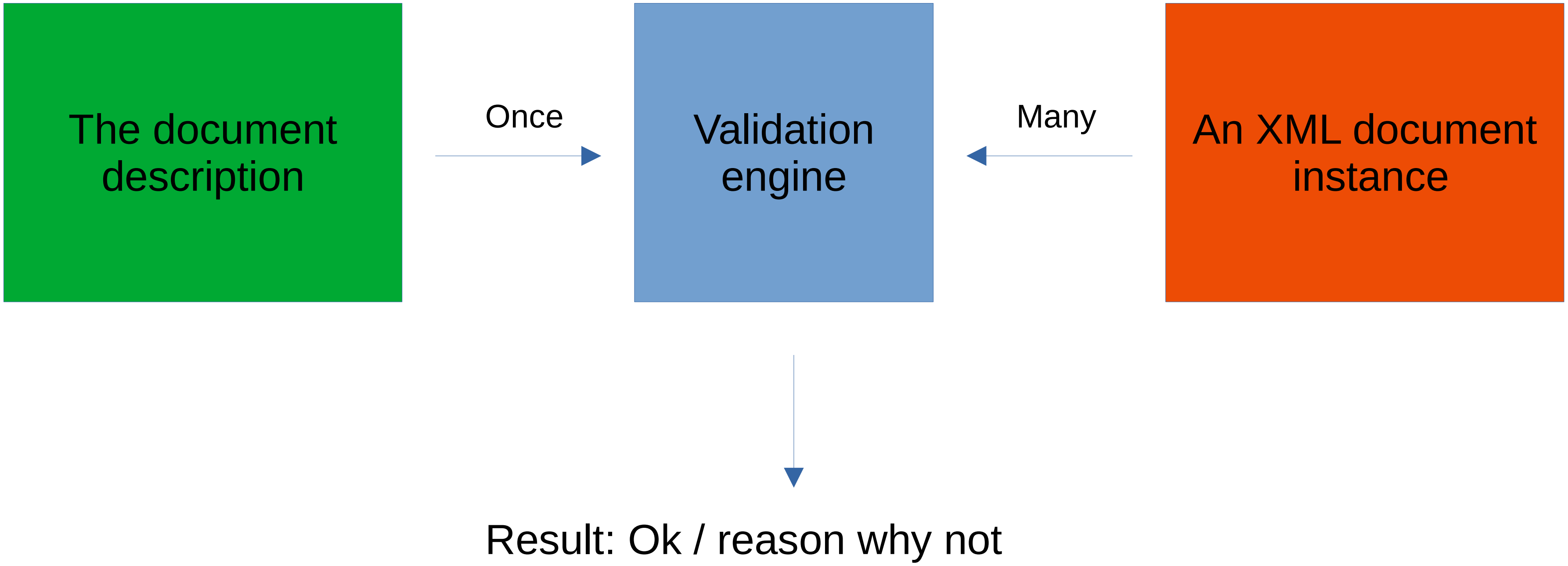
- tDOM schema objects are just Tcl (object) commands, and the „object“ means that every command instance carries its own client data
- Configuring means to tell a schema command the structure of a document type
- Context dependent Tcl commands are basically ordinary Tcl commands (created with Tcl_CreateObjCommand()) that are useful only in a special evaluation context

What is this about?

- Describes a “pattern“ or “method“ for binary coded DSL
  - The XML stuff is just a full flegded example
- Useful especially if the clientData of your Tcl command needs complex and nested configuration
- Think of:

```
createMyComplexCmd myComplexCmdInstance
myComplexCmdInstance configure {
    # an ordinary Tcl script evaluated with a Tcl_Eval*() call
    thisConfigurationCmd $data {
          anotherConfigCmd $otherdata; #nested
    }
    …
}
```

The task: Create a validation engine for tDOM



The document description

Once

Validation engine

Many

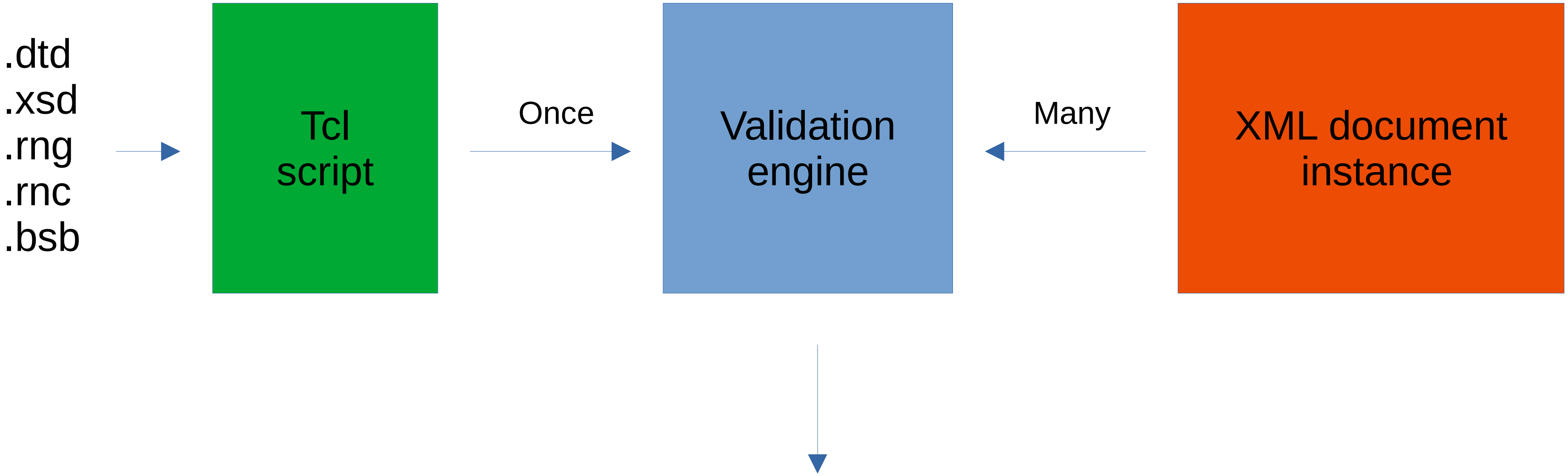
An XML document instance

Result: Ok / reason why not

XML schema languages

- DTD (own syntax)
- XSD, aka XML schema; W3C schema (XML vocabulary)
- Relax NG (XML vocabulary)
- Relax NG compact syntax (own syntax)
- DSD (XML vocabulary)
- Others (including application specific document descriptions)

How about:



.dtd
.xsd
.rng
.rnc
.bsb

Tcl script

Once

Validation engine

Many

XML document instance

Result: Ok / reason why not

The validating part

- There is only a limited, small set of structural content constraints: Sequence, choice, any and variants and combinations of them
- There are academic studies about the algorithms and prior art
- I did it already two times, in Tcl and C

⇨ I knew how to start this part

So I started with the validation part

- Postponed development of XML schema language reader/parser
- But to test the engine I needed to fill the structure description related parts of my validation command clientData
- For this Tcl and his C API was invented!
- Wrote Tcl commands to fill up my validation clientData structure
- Realized only after the fact what I do

How does it look like?

Relax NG example

```
<element name="addressBook"
 xmlns="http://relaxng.org/ns/structure/1.0">
  <zeroOrMore>
    <element name="card">
      <element name="name">
        <text/>
      </element>
      <element name="email">
        <text/>
      </element>
    </element>
  </zeroOrMore>
</element>
```

The same as tDOM schema

```
tdom::schema myschema
myschema define {
    defelement addressBook {
        element card * {
            element name text
            element email text
        }
    }
}
```

Schema definition script overview

Type defines:

```
defelement
defelementtype
defpattern
deftexttype
```

Special ones:

```
start
prefixns
tcl
```

Constraint command:

```
element
attribute, nsattribute
choice
interleave
group
mixed
text
any
ref
namespace
```

… and some miscellaneous

Text constraint scripts

Text may have constraints:

```
element nr {
  text {
    oneOf {
      fixed “undefined“
      positiveInteger
    }
  }
}
```

Or define text types for reuse:

```
deftext nrtype {
  oneOf {
    fixed “undefined“
    positiveInteger
  }
}

element nr {
  text type nrtype
}
```

Text type commands (examples)

Basic type tests:

```
integer, positivInteger,
negativInteger,
nonNegativInteger etc.,
number, boolean, date,
base64, enumeration, ….
```

Logical constructs:

```
allOf
oneOf
not
```

Text properties tests:

```
length, maxlength, minlength
match
regexp
id, idref, key, keyref, ...
```

```
Processed text value:
```

```
whitespace <script>
split ?type ?args?? <script>
```

And most important: `tcl tclcmd ?arg arg ...?` => anything Tcl scriptable

How to use validation commands

- Create and configure them:

```
tdom::schema myschema
myschema define {…<definition script>...}
```

- Standalone for document instance validation from string, filename or channel:

```
set result [myschema validate $somexml]
set result [myschema validatefile $filename]
set result [myschema validatechannel $chan]
```

- By signaling consecutive events to the validation engine:

```
myschema event start someElement
myschema event text “the text content“
myschema event end      ;# ends the current innermost level
```

How to use validation commands (continued)

- On the fly validation while parsing XML into a DOM tree:

  ```
  set doc [dom parse -validateCmd myschema $xmldata
  ```

- On the fly validation while SAX parsing XML:

  ```
  xml::parser myparser -elementstartcommand elmstart \
      -elementendcommand elmend \
      . . . \
      -validateCmd myschema
  myparser parse $xmldata
  ```

- Post validation of DOM trees or subtrees:

  ```
  set doc [dom parse $xml]
  set result [myschema domvalidate $doc]
  ```

Using Tcl scripts for configuration: Pros

- You don’t have to invent a configuration syntax

  Corollary 1: You don’t have to write a parser for this

  Corollary 2: You don’t have to document the syntax

  Corollary 3: Your audience is already familiary with the syntax
- Your configuration language inherits a full-fledged script language

  Especially useful: procs, loops, I/O

Using Tcl scripts for configuration: Pros

```
proc myStandardAttributes {} {
    attribute id ?
    attribute alt ?
    attribute style ?
}

tdom::schema myschema

myschema defelement doc {
    element header {
        myStandardAttributes
        element supplier {
            myStandardAttributes
            text
        }
    }
}
```

```
tdom::schema myschema

set parts {part1 part2 part3}

foreach schemaPart $parts {
    set fd [open $schemaPart]
    myschema define [read $fd]
    close $fd
}
```

Using Tcl scripts for configuration: Cons

- In case of legacy or standard configuration formats: Obiviously you need a script to convert that format into a Tcl configuration script
- Risk of task related implementation: only the 60% features of the legacy format needed for the task gets implemented, not the fully standard
- Not a problem if you use the method to configure or manipulate your hierarchical data because of the elegance and clearness of the code pattern
- Other example in tDOM: `appendFromScript` (and friends)

```
# node cmd creation omitted
dom createDocument myDoc doc
set root [$doc documentElement]
[$doc documentElement] appendFromScript {
    foo {
        bar {text “some content“}
    }
    grill {text “more data“}
}
puts [$doc asXML]
```

```
<myDoc>
    <foo>
        <bar>some content</bar>
    </foo>
    <grill>more data</grill>
</myDoc>
```

Implementation details: Three things are needed

- A place to store a pointer
- The “master“ command which needs complex configuration

```
sdata = initSchemaData (); /* Mallocs and inits complex structure */
Tcl_CreateObjCommand (interp, cmdName, instanceCmd, (ClientData) sdata,
                      instanceDelete);
```

- The context sensitive commands to configure (typically several). They have no own clientData, but work on the clientData of the master.
- … and, well, a bit care with the C implementation (explained below)

A place to store a pointer

- Use a block of thread-private data (“thread global“)
- This can used savely because of the “Appartment model“ – one Tcl interp per thread

```
static Tcl_ThreadDataKey activeSchemaData;
# define GETASI  *(SchemaData**) Tcl_GetThreadData(&activeSchemaData, \
                                                     sizeof(SchemaData*))
static void SetActiveSchemaData (SchemaData *v)
{
    SchemaData **schemaInfoPtr =
        Tcl_GetThreadData(&activeSchemaData,
                          sizeof (SchemaData*));
    *schemaInfoPtr = v;
}
# define SETASI(v) SetActiveSchemaData (v)
```

The master command

- Typically a complex command with one method (of several) to evaluate configuration

```
int instancCmd (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *const objv[]) {
    myCompexType sdata = (myComplexTpye *)clientData;
    ...
    if (Tcl_GetIndexFromObj (interp, objv[1], methods, "method", 0, &mindex) != TCL_OK) {
            return TCL_ERROR;
    }
    switch (mindex) {
    …
    m_configure:
        savedglobal = GETASI;
        if (savedglobal == clientData) {/* Check/handle/error out for recurive call */}
        /* other preperation/checks/stuff */
        sdata->evalStub[3] = objv[2];
        SETASI(sdata);
        sdata→currentEvals++;
        result = Tcl_EvalObjv (interp, 4, sdata->evalStub, TCL_EVAL_GLOBAL);
        sdata→currentEvals--;
        SETASI(savedglobal);
        /* Handle result */
    }
}
```

The master command (continued)

- In an evaluted Tcl script everything can happen

```
tdom::schema mySchema
mySchema define {
    defElement {
        header {
              mySchema define {…}; # Recursive call
        }
        products {
             mySchema delete; # Deletes the command which Tcl_Eval() the script
        }
    }
}
```

- Command destroy needs to check the evaluation counter

The master command (continued)

- The command destroy function may have to postpone the actual cleanup

```
static void instanceDelete (ClientData clientData) {
    SchemaData *sdata = (SchemaData *) clientData;
    if (sdata->currentEvals > 0) {
        sdata->cleanupAfterUse = 1;
        return;
    }
    /* The actual cleanup / freeing memory
}
```

- The instance implementation function have to check for postphoned delete

```
if (sdata->cleanupAfterUse && sdata->currentEvals == 0) {
    instanceDelete (sdata);
}
```

- Every Tcl_Eval*() using method of the master command has to check this

Context sensitive commands

- Most time it is recommende to place them in an own namespace

```
Tcl_CreateObjCommand (interp, "tdom::schema::element",
                     ElementPatternObjCmd, NULL, NULL);
```

- The command looks up the schema data to work on with the macro from above

```
static int ElementPatternObjCmd (…) {
    ...
    SchemaData *sdata = GETASI;
    …
    if (!sdata) {
        SetResult ("Command called outside of schema context");
        return TCL_ERROR;
    }
    …
}
```

Context sensitive commands (continued)

- If the context sensitive command itself evaluates a script, care about the nested evaluation count:

```
static int ElementPatternObjCmd (…) {
    ...
    SchemaData *sdata = GETASI;
    …
    sdata->evalStub[3] = objv[scriptIndex];
    sdata→currentEvals++;
    result = Tcl_EvalObjv (interp, 4, sdata->evalStub, TCL_EVAL_GLOBAL);
    sdata→currentEvals--;
    /* Handle result */
}
```

That’s it!

Questions? Comments?