



Effective **X**ML Communication Using **S**OAP and **A**utomata

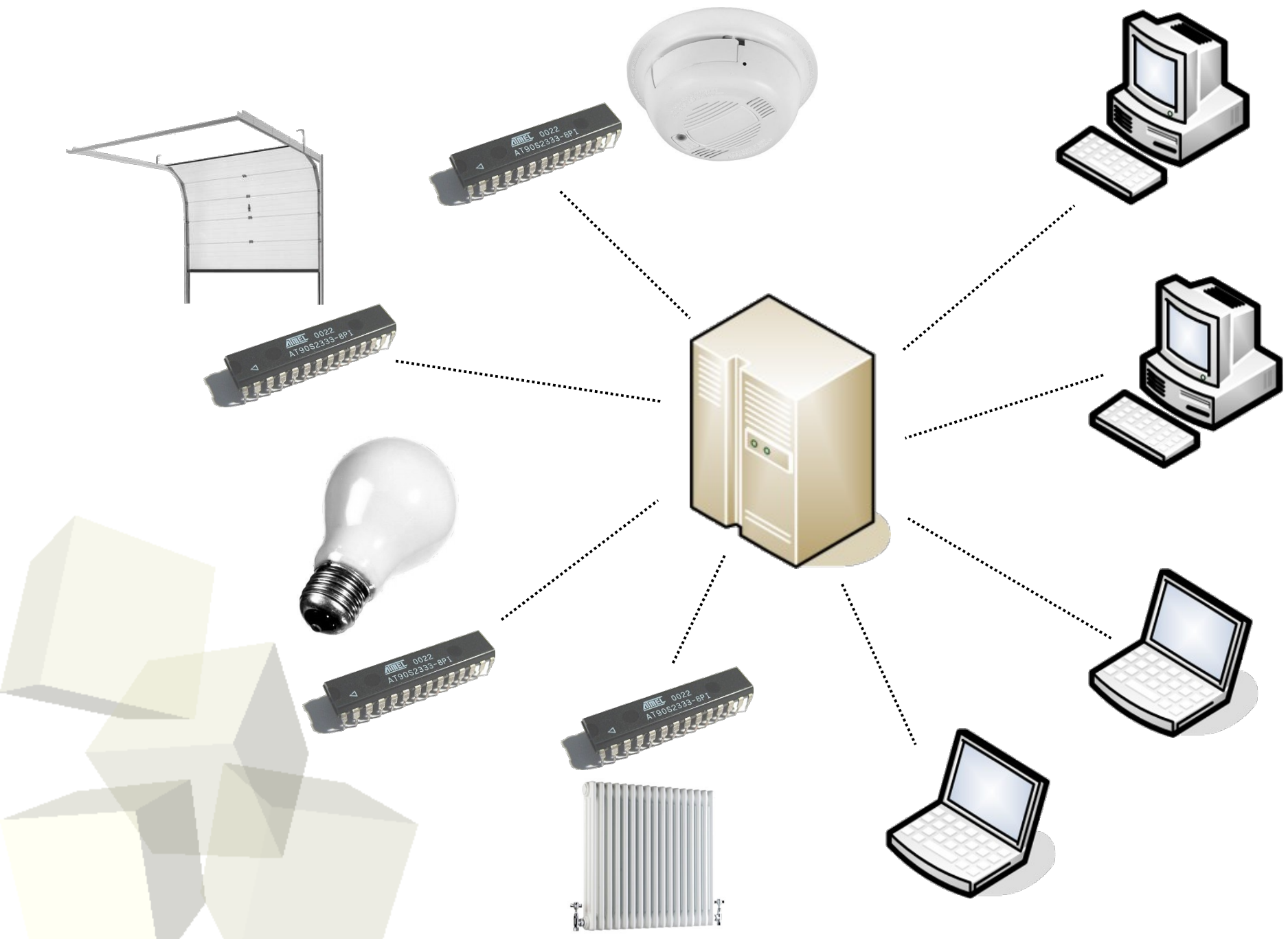
Pavol Rusnák

Supervisor: Doc. Ing. Jan Janeček, CSc.





Motivation





SOAP Message

```
<Envelope>
  <Body>
    <status>
      <fire>false</fire>
      <light>4</light>
      <temp>23.4</temp>
      <temp>18.3</temp>
      <temp>20.7</temp>
      <client>leira</client>
      <client>spectra</client>
      ...
    </status>
  </Body>
</Envelope>
```



Web Service Definition

```
<definitions>
  <types>
    <element name="status">
      <complexType>
        <sequence>
          <element name="fire" type="s:boolean" />
          <element name="light" type="s:integer" />
          <element name="temp" type="s:float"
            minOccurs="3" maxOccurs="3" />
          <element name="client" type="s:string"
            minOccurs="0" maxOccurs="unbounded" />
        </sequence>
      </complexType>
    </element>
  </types>

  <message name="GetStatus">
    <part name="body" element="status"/>
  </message>
</definitions>
```



$S \rightarrow \langle \text{Envelope} \rangle A \langle / \text{Envelope} \rangle$
 $A \rightarrow \langle \text{Body} \rangle B \langle / \text{Body} \rangle$
 $B \rightarrow \langle \text{status} \rangle C \langle / \text{status} \rangle$
 $C \rightarrow \mathbf{DFHK}$
 $D \rightarrow \langle \text{fire} \rangle E \langle / \text{fire} \rangle$
 $E \rightarrow \{ \textit{boolean} \}$
 $F \rightarrow \langle \text{light} \rangle G \langle / \text{light} \rangle$
 $G \rightarrow \{ \textit{integer} \}$
 $H \rightarrow \mathbf{III}$
 $I \rightarrow \langle \text{temp} \rangle J \langle / \text{temp} \rangle$
 $J \rightarrow \{ \textit{float} \}$
 $K \rightarrow \mathbf{LK}$
 $K \rightarrow \varepsilon$
 $L \rightarrow \langle \text{client} \rangle M \langle / \text{client} \rangle$
 $M \rightarrow \{ \textit{string} \}$



S → <Envelope> **A** </Envelope>

A → <Body> **B** </Body>

B → <status> **C** </status>

C → **DFHK**

D → <fire> **E** </fire>

E → {*boolean*}

F → <light> **G** </light>

G → {*integer*}

H → **III**

I → <temp> **J** </temp>

J → {*float*}

K → **LK**

K → ϵ

L → <client> **M** </client>

M → {*string*}

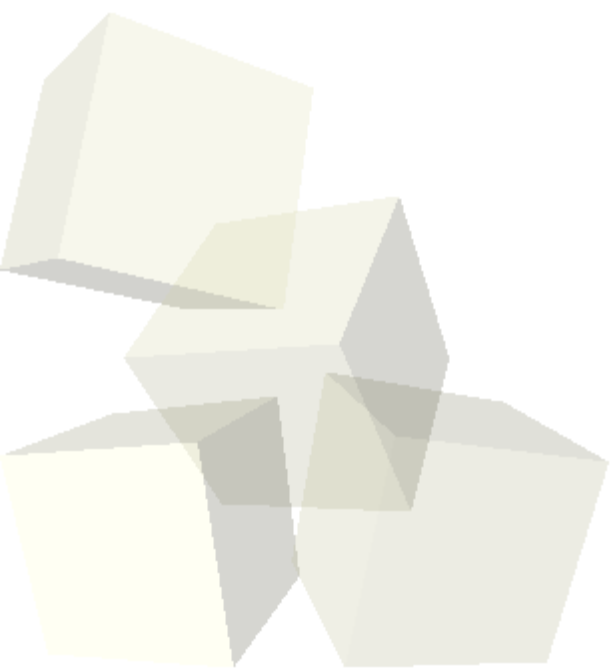
rule types:

- tag
- substitution
- value type



`<element name="tag" type="s:integer" />`

$A \rightarrow \langle \text{tag} \rangle B \langle / \text{tag} \rangle$
 $B \rightarrow \{ \textit{integer} \}$





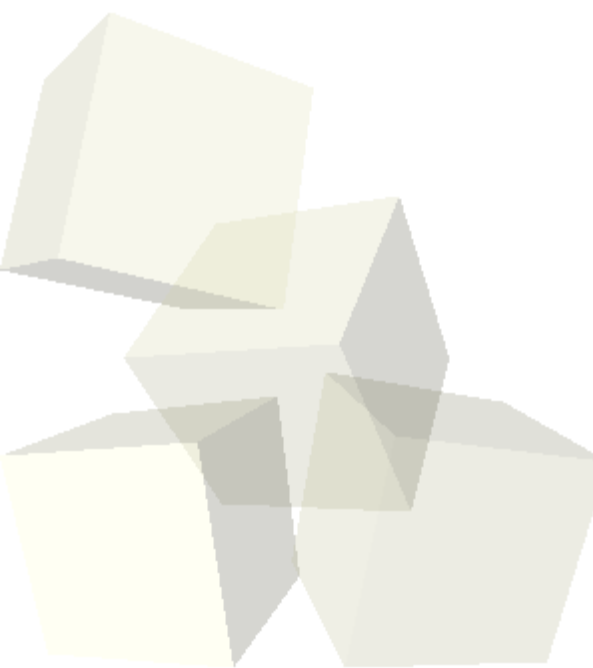
Optional Element

```
<element name="tag" type="s:integer"
  minOccurs="0" />
```

$A \rightarrow \langle \text{tag} \rangle B \langle / \text{tag} \rangle$

$A \rightarrow \varepsilon$

$B \rightarrow \{integer\}$





Repeating Element

```
<element name="tag" type="s:integer"  
  minOccurs="1" maxOccurs="3" />
```

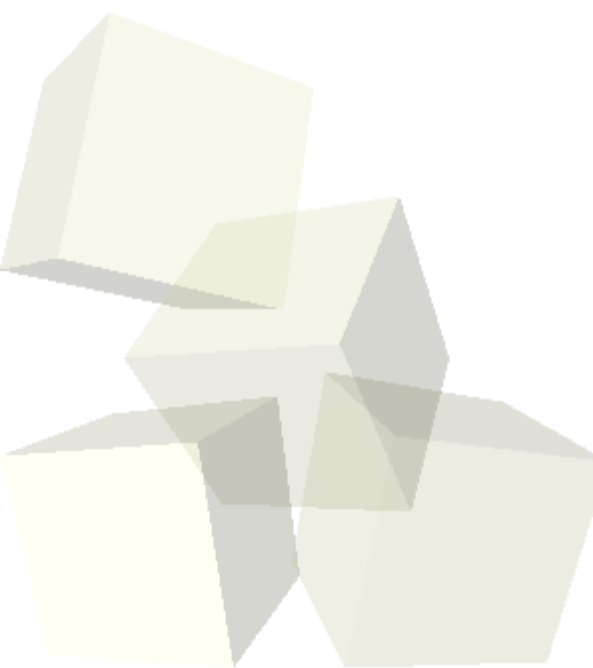
A → *BBB*

A → *BB*

A → *B*

B → <tag> *C* </tag>

C → {*integer*}





Infinitely Repeating Element

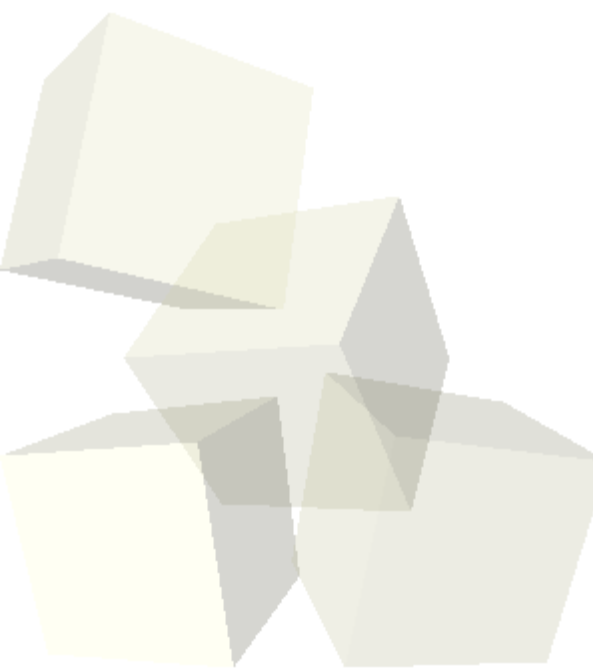
```
<element name="tag" type="s:integer"  
  minOccurs="0" maxOccurs="unbounded"/>
```

$A \rightarrow BA$

$A \rightarrow \varepsilon$

$B \rightarrow \langle \text{tag} \rangle C \langle / \text{tag} \rangle$

$C \rightarrow \{integer\}$





Complex Type: Sequence

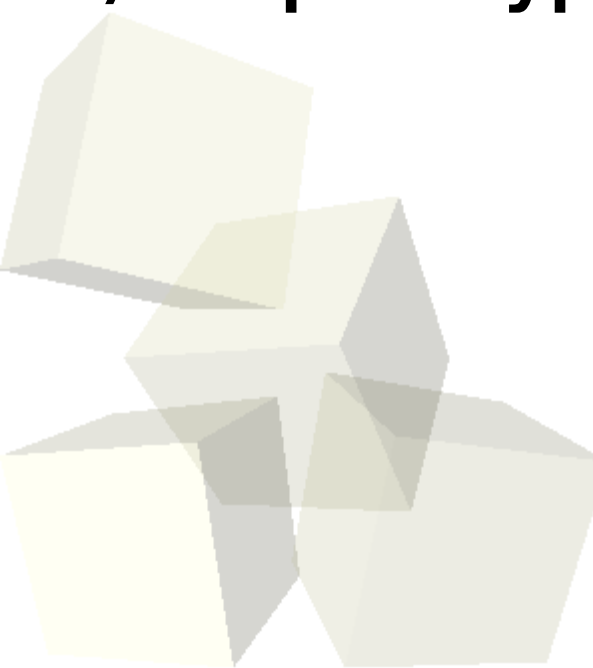
```
<complexType>  
  <sequence>  
    <element name="tag1" type="s:integer" />  
    <element name="tag2" type="s:integer" />  
    <element name="tag3" type="s:integer" />  
  </sequence>  
</complexType>
```

A → **BCD**

B → <tag1> **E** </tag1>

C → <tag2> **F** </tag2>

D → <tag3> **G** </tag3>





Complex Type: Choice

```
<complexType>  
  <choice>  
    <element name="tag1" type="s:integer" />  
    <element name="tag2" type="s:integer" />  
    <element name="tag3" type="s:integer" />  
  </choice>  
</complexType>
```

A → *B*

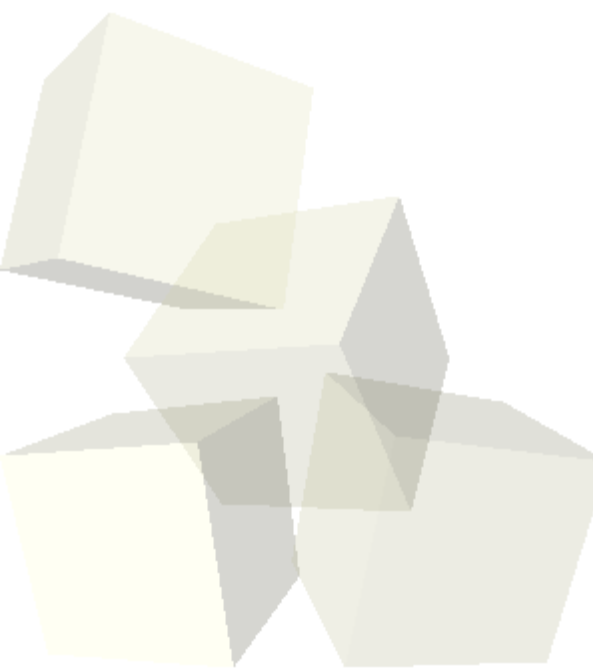
A → *C*

A → *D*

B → <tag1> *E* </tag1>

C → <tag2> *F* </tag2>

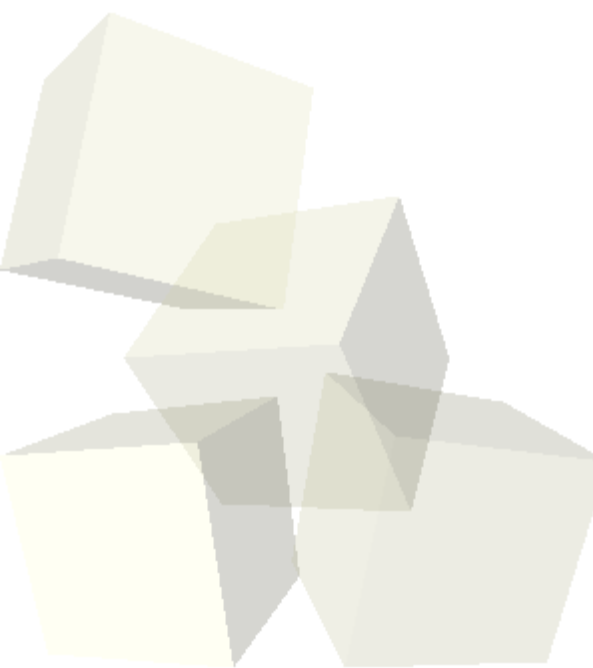
D → <tag3> *G* </tag3>





Complex Type: All

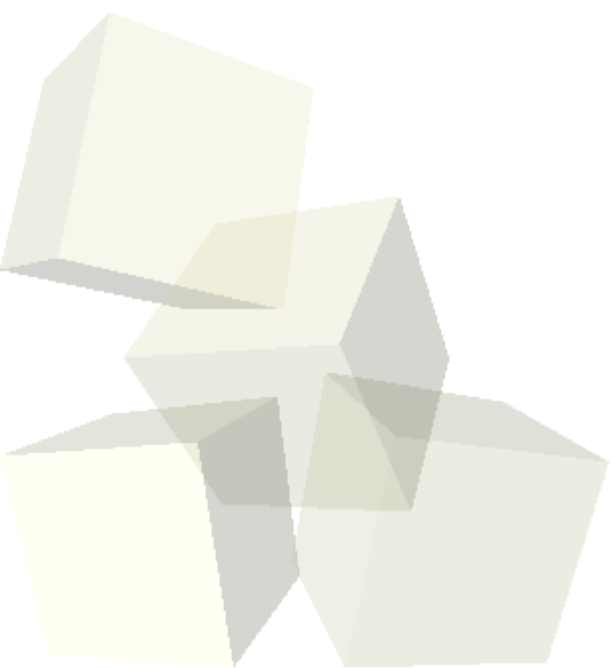
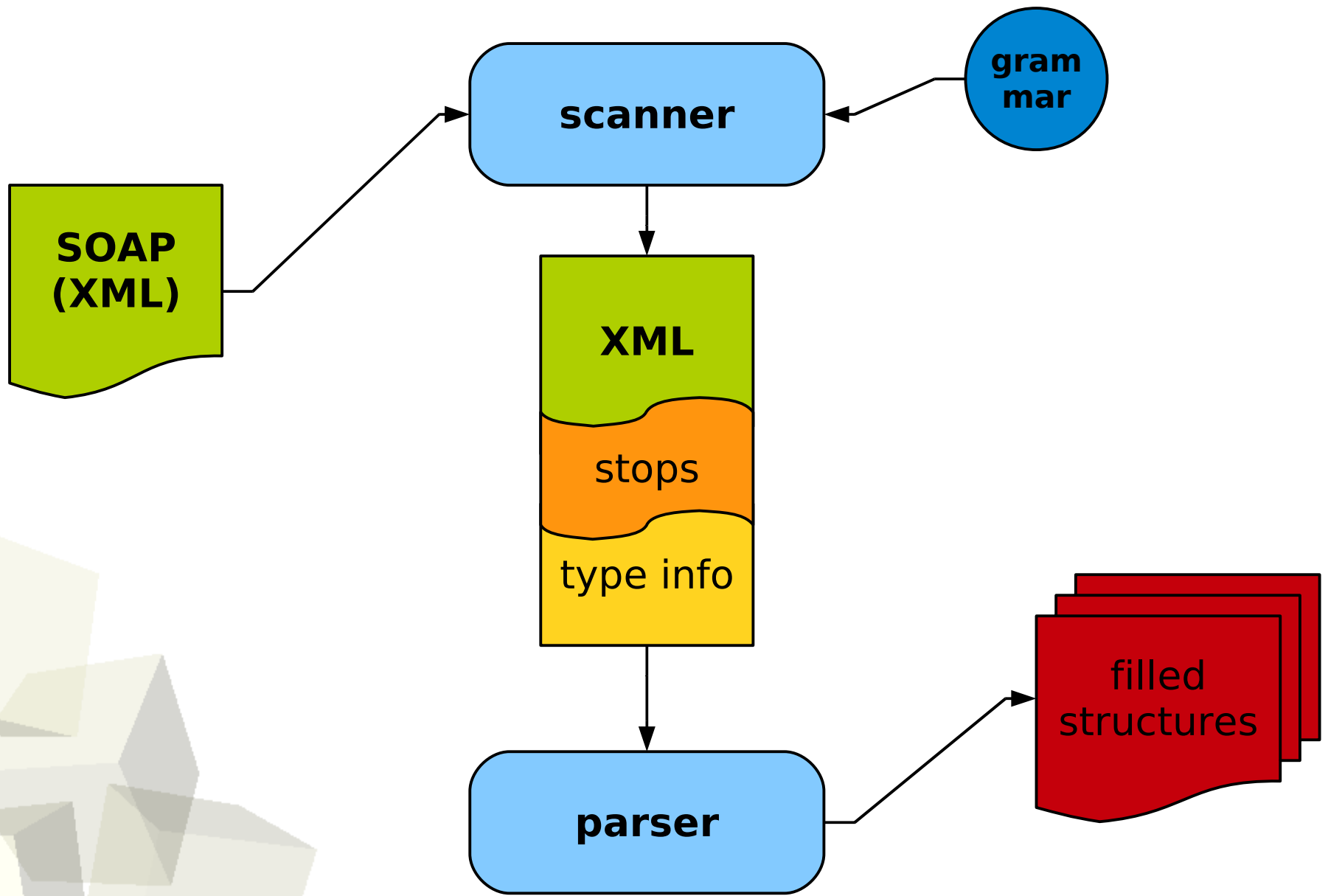
```
<complexType>  
  <all>  
    <element name="tag1" type="s:integer" />  
    <element name="tag2" type="s:integer" />  
    <element name="tag3" type="s:integer" />  
  </all>  
</complexType>
```



A → *BCD*
A → *BDC*
A → *CBD*
A → *CDB*
A → *DBC*
A → *DCB*



Deserialization Schema





Deserialization Stops

<Envelope>

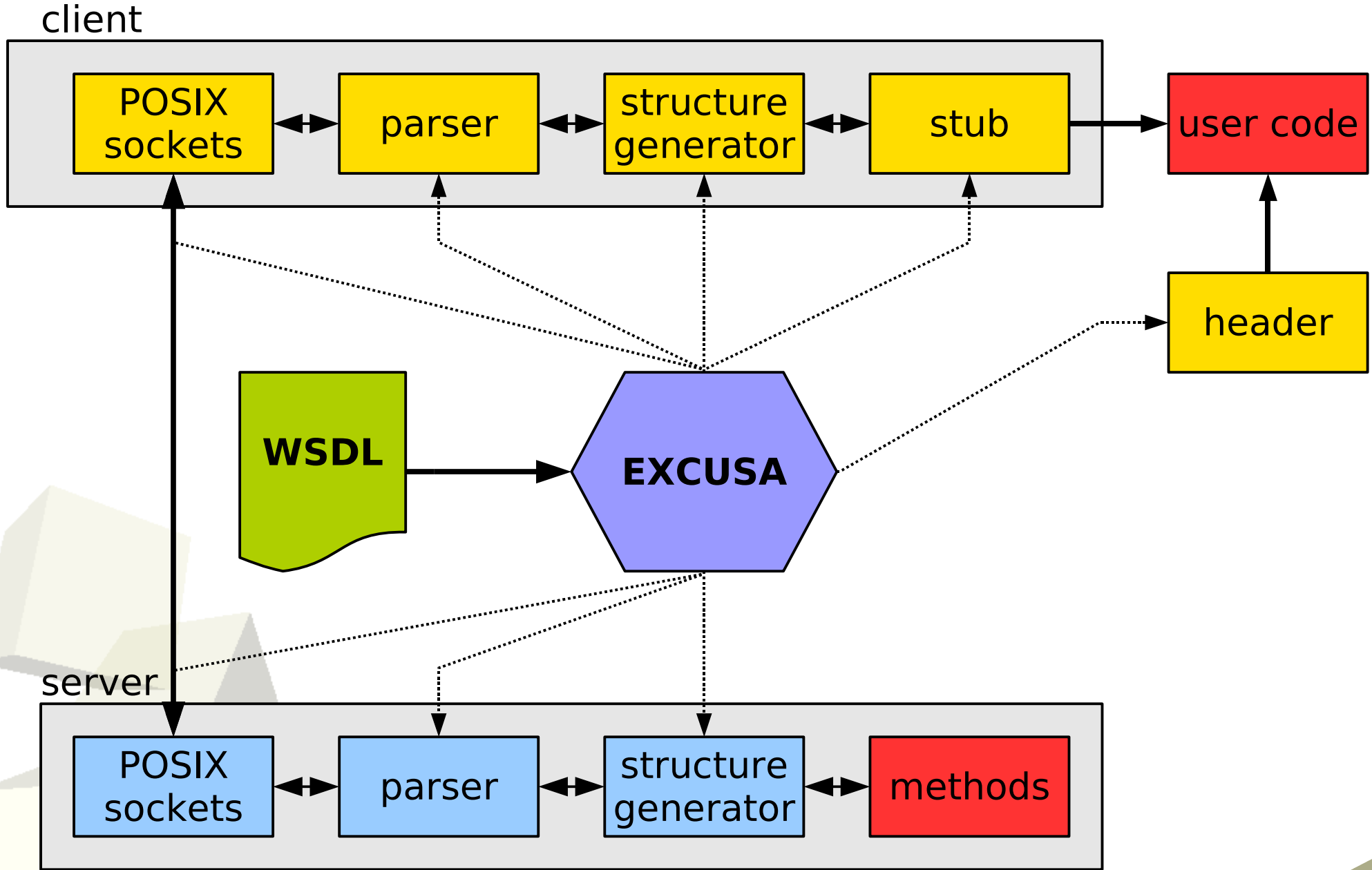
<Body>

```
<status>  
  <fire> false /fire>  
  <light> 4 /light>  
  <temp> 23.4 /temp>  
  <temp> 18.3 /temp>  
  <temp> 20.7 /temp>  
  <client> leira /client>  
  <client> spectra /client>  
  ...  
</status>
```

</Body>

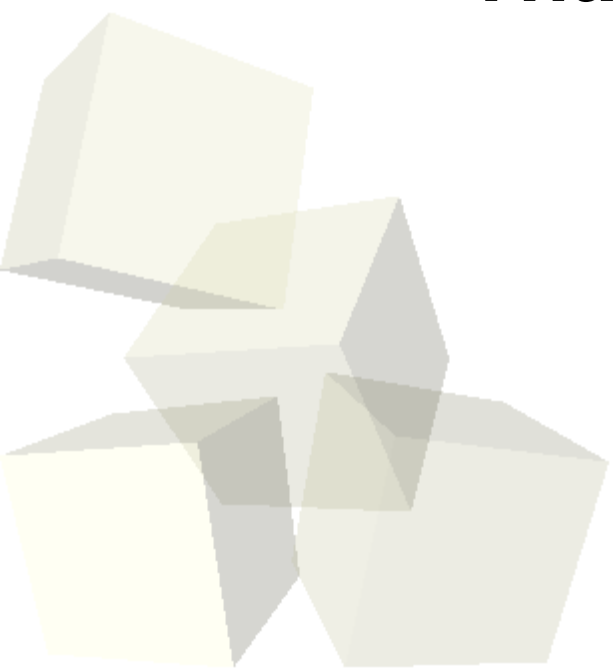
</Envelope>

Solution Schema





Thank you for your attention!





XML Grammar Representation

```
int msg0[] = {0, tag_Envelope, 1}; // S → <Envelope> A </Envelope>
int msg1[] = {0, tag_Body, 2}; // A → <Body> B </Body>
int msg2[] = {0, tag_status, 3}; // B → <status> C </status>
int msg3[] = {1, 4, 6, 8, 11+1«SHIFT, -1}; // C → DFHK
int msg4[] = {0, tag_fire, 5}; // D → <fire> E </fire>
int msg5[] = {-7}; // E → {boolean}
int msg6[] = {0, tag_light, 7}; // F → <light> G </light>
int msg7[] = {-2}; // G → {integer}
int msg8[] = {1, 9, 9, 9, -1}; // H → III
int msg9[] = {0, tag_temp, 10}; // I → <temp> J </temp>
int msg10[] = {-4}; // J → {float}
int msg11[] = {1, 13, 11+1«SHIFT, -1}; // K → LK
int msg12[] = {1, -1}; // K → ε
int msg13[] = {0, tag_client, 14}; // L → <client> M </client>
int msg14[] = {-1}; // M → {string}
```

```
int *msg_grammar[] = { msg0, msg1, msg2, msg3, msg4, msg5, msg6, msg7,
                      msg8, msg9, msg10, msg11, msg12, msg13, msg14};
```

```
enum msg_tags { tag_Envelope, tag_Body, tag_status, tag_fire,
               tag_light, tag_temp, tag_client };
```

```
char *msg_tags_str = { "Envelope", "Body", "status", "fire",
                      "light", "temp", "client" };
```



XML Grammar Representation

```
int msg0[] = {0, tag_Envelope, 1}; // S → <Envelope> A </Envelope>
int msg1[] = {0, tag_Body, 2}; // A → <Body> B </Body>
int msg2[] = {0, tag_status, 3}; // B → <status> C </status>
int msg3[] = {1, 4, 6, 8, 11+1«SHIFT, -1}; // C → DFHK
int msg4[] = {0, tag_fire, 5}; // D → <fire> E </fire>
int msg5[] = {-7}; // E → {boolean}
int msg6[] = {0, tag_light, 7}; // F → <light> G </light>
int msg7[] = {-2}; // G → {integer}
int msg8[] = {1, 9, 9, 9, -1}; // H → III
int msg9[] = {0, tag_temp, 10}; // I → <temp> J </temp>
int msg10[] = {-4}; // J → {float}
int msg11[] = {1, 13, 11+1«SHIFT, -1}; // K → LK
int msg12[] = {1, -1}; // K → ε
int msg13[] = {0, tag_client, 14}; // L → <client> M </client>
int msg14[] = {-1}; // M → {string}

int *msg_grammar[] = { msg0, msg1, msg2, msg3, msg4, msg5, msg6, msg7,
                      msg8, msg9, msg10, msg11, msg12, msg13, msg14};

enum msg_tags { tag_Envelope, tag_Body, tag_status, tag_fire,
               tag_light, tag_temp, tag_client };

char *msg_tags_str = { "Envelope", "Body", "status", "fire",
                      "light", "temp", "client" };
```



XML Grammar Representation

```
int msg0[] = {0, tag_Envelope, 1}; // S → <Envelope> A </Envelope>
int msg1[] = {0, tag_Body, 2}; // A → <Body> B </Body>
int msg2[] = {0, tag_status, 3}; // B → <status> C </status>
int msg3[] = {1, 4, 6, 8, 11+1«SHIFT, -1}; // C → DFHK
int msg4[] = {0, tag_fire, 5}; // D → <fire> E </fire>
int msg5[] = {-7}; // E → {boolean}
int msg6[] = {0, tag_light, 7}; // F → <light> G </light>
int msg7[] = {-2}; // G → {integer}
int msg8[] = {1, 9, 9, 9, -1}; // H → III
int msg9[] = {0, tag_temp, 10}; // I → <temp> J </temp>
int msg10[] = {-4}; // J → {float}
int msg11[] = {1, 13, 11+1«SHIFT, -1}; // K → LK
int msg12[] = {1, -1}; // K → ε
int msg13[] = {0, tag_client, 14}; // L → <client> M </client>
int msg14[] = {-1}; // M → {string}

int *msg_grammar[] = { msg0, msg1, msg2, msg3, msg4, msg5, msg6, msg7,
                      msg8, msg9, msg10, msg11, msg12, msg13, msg14};

enum msg_tags { tag_Envelope, tag_Body, tag_status, tag_fire,
               tag_light, tag_temp, tag_client };

char *msg_tags_str = { "Envelope", "Body", "status", "fire",
                      "light", "temp", "client" };

```



XML Grammar Representation

```
int msg0[] = {0, tag_Envelope, 1}; // S → <Envelope> A </Envelope>
int msg1[] = {0, tag_Body, 2}; // A → <Body> B </Body>
int msg2[] = {0, tag_status, 3}; // B → <status> C </status>
int msg3[] = {1, 4, 6, 8, 11+1«SHIFT, -1}; // C → DFHK
int msg4[] = {0, tag_fire, 5}; // D → <fire> E </fire>
int msg5[] = {-7}; // E → {boolean}
int msg6[] = {0, tag_light, 7}; // F → <light> G </light>
int msg7[] = {-2}; // G → {integer}
int msg8[] = {1, 9, 9, 9, -1}; // H → III
int msg9[] = {0, tag_temp, 10}; // I → <temp> J </temp>
int msg10[] = {-4}; // J → {float}
int msg11[] = {1, 13, 11+1«SHIFT, -1}; // K → LK
int msg12[] = {1, -1}; // K → ε
int msg13[] = {0, tag_client, 14}; // L → <client> M </client>
int msg14[] = {-1}; // M → {string}
```

```
int *msg_grammar[] = { msg0, msg1, msg2, msg3, msg4, msg5, msg6, msg7,
                      msg8, msg9, msg10, msg11, msg12, msg13, msg14};
```

```
enum msg_tags { tag_Envelope, tag_Body, tag_status, tag_fire,
               tag_light, tag_temp, tag_client };
```

```
char *msg_tags_str = { "Envelope", "Body", "status", "fire",
                      "light", "temp", "client" };
```