Constraining XML instances using standard logic properties

When I was in school I learned about the transitive property, symmetric property, and others. They seemed so abstract and irrelevant to anything in the real world. Certainly not relevant to XML.

So I quickly forgot them.

Recently I’ve come to realize that these properties are relevant. Some smart people identified and defined them because they occur so frequently in the real world! They are super-relevant to XML.

Wow, what a wake-up call for me.

In this message I will demonstrate the relevance of these 7 standard properties: transitive, irreflexive, symmetric, functional, injective, total, and onto.

Below you will see that XML Schema is not powerful enough to constrain XML instances to satisfy these properties. Schematron must be used.

Question: Have you ever designed XML documents to satisfy one or more of these properties?

Transitive Property

Often I create XML documents containing pairs of values. For example, below is an XML document containing three pairs, representing a sibling relationship: John has a sibling Tom, Tom has a sibling Barb, and John has (another) sibling Barb:

<Siblings>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>Tom</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>Tom</Person1>  
 <Person2>Barb</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>Barb</Person2>  
 </Sibling>  
</Siblings>

An XML Schema might be created to express this: Instance documents contain any number of sibling pairs. According to the schema, then, this would be a valid instance:

<Siblings>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>Tom</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>Tom</Person1>  
 <Person2>Barb</Person2>  
 </Sibling>  
</Siblings>

However, from our knowledge of the sibling relation we know that it has this property: If A and B are siblings and B and C are siblings, then A and C are siblings. That’s the transitive property!

So the second XML document is not a valid instance of the sibling relation – John and Tom are siblings and Tom and Barb are siblings, so John and Barb should be siblings. The transitive property does not hold in the document.

XSD 1.0 does not have the power to express the constraint to ensure that XML instances satisfy the transitive property. To express the constraint, use Schematron.

An XML document has the transitive property if it satisfies this constraint:

<sch:pattern id="transitive">  
 <sch:rule context="Sibling">  
 <sch:let name="from" value="Person1"/>  
 <sch:let name="to" value="Person2"/>  
 <sch:assert test="if (not(../Sibling[Person1 eq $to])) then true()  
 else every $j in  
 (for $i in ../Sibling[Person1 eq $to] return  
 if (../Sibling[(Person1 eq $from) and (Person2 eq $i/Person2)]) then true()  
 else false())  
 satisfies $j eq true()">  
 The transitive property fails to hold   
 for this sibling.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Neat!

Alloy Version

The following signature (**sig**) declarations creates the sibling relation:

**abstract** **sig** Person {

sibling: **set** Person

}

**one** **sig** John **extends** Person {}

**one** **sig** Tom **extends** Person {}

**one** **sig** Barb **extends** Person {}

Those signature declarations are the Alloy analog to sibling.xsd shown at the bottom of this message.

This Alloy **fact** creates a specific instance of the sibling relation:

**fact** instance {

sibling =

John -> Tom +

Tom -> Barb +

John -> Barb

}

That is the Alloy analog to the first XML instance shown above.

The following **fact** constrains the sibling relation so that it has the transitive property:

**fact** transitive {  
 sibling.sibling **in** sibling  
}

That is the Alloy analog to the Schematron shown above.

Sidebar: Terminology

1. The set of pairs is a binary relation.

2. Schematron is used to constrain the relation.

3. The constrained relation has the transitive property.

4. The transitive property is one of the standard properties.

5. An XML instance has the transitive property if it satisfies the constraint.

Irreflexive Property

Continuing with the sibling example…

A person cannot be a sibling of himself/herself. So this XML instance is invalid:

<Siblings>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>John</Person2>  
 </Sibling>  
</Siblings>

John cannot be a sibling of himself.

From our knowledge of the sibling relation we know that it possesses this property: A person cannot be a sibling of himself/herself. (No identity pairs.) That property is called the irreflexive property.

An XML document has the irreflexive property if it satisfies this constraint:

<sch:pattern id="irreflexive">  
 <sch:rule context="Sibling">  
 <sch:assert test="Person1 ne Person2">  
 The irreflexive property fails to hold   
 for this sibling.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

The following **fact** constrains the sibling relation so that it has the irreflexive property:

**fact** irreflexive {  
 sibling.sibling **in** sibling  
}

That is the Alloy analog to the Schematron shown above.

Symmetric Property

The sibling relation is symmetric: If John has a sibling Tom, then Tom has a sibling John.

The symmetric property holds in this instance:

<Siblings>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>Tom</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>Tom</Person1>  
 <Person2>John</Person2>  
 </Sibling>  
</Siblings>

The instance satisfies the symmetric property but not the transitive property. Can you create an instance that satisfies both the symmetric and transitive properties? Should XML instances be required to satisfy both properties, or just one?

An XML document has the symmetric property if it satisfies this constraint:

<sch:pattern id="symmetric">  
 <sch:rule context="Sibling">  
 <sch:let name="from" value="Person1"/>  
 <sch:let name="to" value="Person2"/>  
 <sch:assert test="some $i in ../Sibling satisfies ($i/Person1 eq $to) and ($i/Person2 eq $from)">  
 The symmetric property fails to hold   
 for this sibling.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

The following **fact** constrains the sibling relation so that it has the symmetric property:

**fact** symmetric {  
 ~sibling **in** sibling  
}

That is the Alloy analog to the Schematron shown above.

Transitive, Irreflexive, and Symmetric?

Should XML instances satisfy all three properties? *Can* XML instances satisfy all three properties?

XML instances can satisfy both the transitive and irreflexive properties, as the very first XML instance showed:

<Siblings>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>Tom</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>Tom</Person1>  
 <Person2>Barb</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>Barb</Person2>  
 </Sibling>  
</Siblings>

That satisfies the transitive and irreflexive properties, but not the symmetric property.

Here is an XML instance that satisfies all three properties:

<Siblings></Siblings>

Ugh.

An instance with no pairs does indeed satisfy the transitive, irreflexive, and symmetric properties. Is that an acceptable instance? Possibly but probably not. So let’s refine the question: Can XML instances satisfy the transitive, irreflexive, and symmetric properties, and be *nonempty*?

Answer: No.

Can XML instances satisfy the transitive and symmetric properties (and be nonempty)?

Answer: Yes. Here’s an example:

<Siblings>  
 <Sibling>  
 <Person1>John</Person1>  
 <Person2>John</Person2>  
 </Sibling>  
 <Sibling>  
 <Person1>Tom</Person1>  
 <Person2>Tom</Person2>  
 </Sibling>  
</Siblings>

However, that’s not desirable for the reasons discussed earlier.

A good solution is to require XML instances to satisfy the transitive, and irreflexive properties. This Schematron applies the necessary constraints:

<sch:pattern id="nonempty">  
 <sch:rule context="Siblings">  
 <sch:assert test="Sibling">  
 The nonempty property fails to hold   
 for this instance.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>  
  
<sch:pattern id="transitive">  
 <sch:rule context="Sibling">  
 <sch:let name="from" value="Person1"/>  
 <sch:let name="to" value="Person2"/>  
 <sch:assert test="if (not(../Sibling[Person1 eq $to])) then true()  
 else every $j in  
 (for $i in ../Sibling[Person1 eq $to] return  
 if (../Sibling[(Person1 eq $from) and (Person2 eq $i/Person2)]) then true()  
 else false())  
 satisfies $j eq true()">  
 The transitive property fails to hold   
 for this sibling.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>  
  
<sch:pattern id="irreflexive">  
 <sch:rule context="Sibling">  
 <sch:assert test="Person1 ne Person2">  
 The irreflexive property fails to hold   
 for this sibling.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

**fact** nonempty {

**some** sibling

}

**fact** transitive {

sibling.sibling **in** sibling

}

**fact** irreflexive {

**no** iden **&** sibling

}

Functional Property

[New example] In a company each employee has an office. Several employees may share an office. Here’s a sample instance:

<Directory>  
 <Workplace>  
 <Employee>Bill</Employee>  
 <Office>OfficeA</Office>  
 </Workplace>  
 <Workplace>  
 <Employee>Sara</Employee>  
 <Office>OfficeB</Office>  
 </Workplace>  
 <Workplace>  
 <Employee>Jim</Employee>  
 <Office>OfficeB</Office>  
 </Workplace>  
</Directory>

Bill has his own office. Sara and Jim share an office.

“Workplace” is a binary relation between Employee and Office.

The following is an invalid instance:

<Directory>  
 <Workplace>  
 <Employee>Bill</Employee>  
 <Office>OfficeA</Office>  
 </Workplace>  
 <Workplace>  
 <Employee>Bill</Employee>  
 <Office>OfficeC</Office>  
 </Workplace>  
 <Workplace>  
 <Employee>Sara</Employee>  
 <Office>OfficeB</Office>  
 </Workplace>  
 <Workplace>  
 <Employee>Jim</Employee>  
 <Office>OfficeB</Office>  
 </Workplace>  
</Directory>

Bill occupies two offices – OfficeA and OfficeC – which is not allowed. Valid instances must satisfy this property: Each employee has exactly one office. That is the functional property.

An XML document has the functional property if it satisfies this constraint:

<sch:pattern id="functional">  
 <sch:rule context="Workplace">  
 <sch:let name="employee" value="Employee"/>  
 <sch:assert test="not(preceding-sibling::Workplace[Employee eq $employee]) and   
 not(following-sibling::Workplace[Employee eq $employee])">  
 The functional property fails to hold   
 for this workplace.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

The following signature (**sig**) declarations creates the workplace relation:

**abstract** **sig** Employee {

workplace: **set** Office

}

**abstract** **sig** Office {}

**one** **sig** Bill **extends** Employee {}

**one** **sig** Sara **extends** Employee {}

**one** **sig** Jim **extends** Employee {}

**one** **sig** OfficeA **extends** Office {}

**one** **sig** OfficeB **extends** Office {}

**one** **sig** OfficeC **extends** Office {}

Those signature declarations are the Alloy analog to the workplace XSD shown at the bottom of this message.

This **fact** creates a specific instance of the workplace relation:

**fact** instance {

workplace =

Bill -> OfficeA +

Sara -> OfficeB +

Jim -> OfficeB

}

That is the Alloy analog to the first XML instance shown above.

The following **fact** constrains the workplace relation so that it has the functional property:

**fact** functional {  
 ~workplace.workplace **in** **iden**  
}

That is the Alloy analog to the Schematron shown above.

Injective Property

[New example] In a company each task/project must be staffed by one employee. An employee may be between tasks, and thereby not working on a task. Here’s a sample instance:

<Tasking>  
 <Staffing>  
 <Employee>Bill</Employee>  
 <Task>Task1</Task>  
 </Staffing>  
 <Staffing>  
 <Employee>Sara</Employee>  
 <Task>Task2</Task>  
 </Staffing>  
 <Staffing>  
 <Employee>Sara</Employee>  
 <Task>Task3</Task>  
 </Staffing>  
</Tasking>

Bill is working on Task1. Sara is working on two tasks – Task2 and Task3. Jim is between tasks and not working on any task. There are no other tasks.

“Staffing” is a binary relation between Employee and Task.

The following is an invalid instance:

<Tasking>   
 <Staffing>  
 <Employee>Bill</Employee>  
 <Task>Task1</Task>  
 </Staffing>  
 <Staffing>  
 <Employee>Sara</Employee>  
 <Task>Task1</Task>  
 </Staffing>  
</Tasking>

Two people are working on Task1. That’s not allowed. Valid instances must satisfy this property: Each task must be worked on by exactly one employee. That is called the injective property.

An XML document has the injective property if it satisfies this constraint:

<sch:pattern id="injective">  
 <sch:rule context="Staffing">  
 <sch:let name="task" value="Task"/>  
 <sch:assert test="not(preceding-sibling::Staffing[Task eq $task]) and  
 not(following-sibling::Staffing[Task eq $task])">  
 The injective property fails to hold   
 for this staffing.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

The following signature (**sig**) declarations creates the staffing relation:

**abstract** **sig** Employee {

staffing: **set** Task

}

**abstract** **sig** Task {}

**one** **sig** Bill **extends** Employee {}

**one** **sig** Sara **extends** Employee {}

**one** **sig** Jim **extends** Employee {}

**one** **sig** Task1 **extends** Task {}

**one** **sig** Task2 **extends** Task {}

**one** **sig** Task3 **extends** Task {}

Those signature declarations are the Alloy analog to the staffing XSD shown at the bottom of this message.

This **fact** creates a specific instance of the staffing relation:

**fact** instance {

staffing =

Bill -> Task1 +

Sara -> Task2 +

Sara -> Task3

}

That is the Alloy analog to the first XML instance shown above.

The following **fact** constrains the staffing relation so that it has the injective property:

**fact** injective {

staffing.~staffing **in** **iden**

}

That is the Alloy analog to the Schematron shown above.

Total Property

Consider nodes in a ring (cyclic) arrangement. A “successor” relation maps a node to its following node. Here is a valid XML instance of a 3-node network:

<Network>  
 <Successor>  
 <Node1>Node1</Node1>  
 <Node2>Node2</Node2>  
 </Successor>  
 <Successor>  
 <Node1>Node2</Node1>  
 <Node2>Node3</Node2>  
 </Successor>  
 <Successor>  
 <Node1>Node3</Node1>  
 <Node2>Node1</Node2>  
 </Successor>  
</Network>

The following XML instance is invalid because it is missing a mapping for Node3:

<Network>  
 <Successor>  
 <Node1>Node1</Node1>  
 <Node2>Node2</Node2>  
 </Successor>  
 <Successor>  
 <Node1>Node2</Node1>  
 <Node2>Node3</Node2>  
 </Successor>  
</Network>

For an XML instance to be valid, it must satisfy this constraint: Each node must map to a node (i.e., each node must have a successor). An instance that satisfies that constraint is said to have the total property.

An XML document has the total property if it satisfies this constraint:

<sch:pattern id="total">  
 <sch:rule context="Network">  
 <sch:assert test="Successor[Node1 eq 'Node1']">  
 The total property fails to hold   
 for Node1.  
 </sch:assert>  
 <sch:assert test="Successor[Node1 eq 'Node2']">  
 The total property fails to hold   
 for Node2.  
 </sch:assert>  
 <sch:assert test="Successor[Node1 eq 'Node3']">  
 The total property fails to hold   
 for Node3.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

The following signature (**sig**) declarations creates the successor relation:

**abstract** **sig** Node {

successor: Node

}

**one** **sig** Node1 **extends** Node {}

**one** **sig** Node2 **extends** Node {}

**one** **sig** Node3 **extends** Node {}

Those signature declarations are the Alloy analog to the successor XSD shown at the bottom of this message.

This **fact** creates a specific instance of the successor relation:

**fact** instance

successor =

Node1 -> Node2 +

Node2 -> Node3 +

Node3 -> Node1

}

That is the Alloy analog to the first XML instance shown above.

The following **fact** constrains the successor relation so that it has the total property:

**fact** total {

**univ** **in** successor.univ

}

That is the Alloy analog to the Schematron shown above.

Onto Property

The previous section showed that every node has a successor. XML instances satisfying this constraint have the total property.

Now consider this constraint: Every node must have a mapping *to it*. XML instances satisfying this constraint have the onto property.

An XML document has the onto property if it satisfies this constraint:

<sch:pattern id="onto">  
 <sch:rule context="Network">  
 <sch:assert test="Predecessor[Node2 eq 'Node1']">  
 The onto property fails to hold   
 for Node1.  
 </sch:assert>  
 <sch:assert test="Predecessor[Node2 eq 'Node2']">  
 The onto property fails to hold   
 for Node2.  
 </sch:assert>  
 <sch:assert test="Predecessor[Node2 eq 'Node3']">  
 The onto property fails to hold   
 for Node3.  
 </sch:assert>  
 </sch:rule>  
</sch:pattern>

Alloy Version

The following **fact** constrains the predecessor relation so that it has the onto property:

**fact** onto {

**univ** **in** univ.predecessor

}

That is the Alloy analog to the Schematron shown above.

Sibling XML Schema

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
   
 <xs:simpleType name="Name">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="John" />  
 <xs:enumeration value="Tom" />  
 <xs:enumeration value="Barb" />  
 </xs:restriction>  
 </xs:simpleType>  
   
 <xs:element name="Siblings">  
 <xs:complexType>  
 <xs:sequence minOccurs="0" maxOccurs="unbounded">  
 <xs:element name="Sibling">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="Person1" type="Name" />  
 <xs:element name="Person2" type="Name" />  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
   
</xs:schema>

Workplace XML Schema

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
   
 <xs:simpleType name="Name">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="Bill" />  
 <xs:enumeration value="Sara" />  
 <xs:enumeration value="Jim" />  
 </xs:restriction>  
 </xs:simpleType>  
   
 <xs:simpleType name="Suite">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="Office A" />  
 <xs:enumeration value="Office B" />  
 </xs:restriction>  
 </xs:simpleType>  
   
 <xs:element name="Directory">  
 <xs:complexType>  
 <xs:sequence minOccurs="0" maxOccurs="unbounded">  
 <xs:element name="Workplace">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="Employee" type="Name" />  
 <xs:element name="Office" type="Suite" />  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
   
</xs:schema>

Staffing XML Schema

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
   
 <xs:simpleType name="Name">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="Bill" />  
 <xs:enumeration value="Sara" />  
 <xs:enumeration value="Jim" />  
 </xs:restriction>  
 </xs:simpleType>  
   
 <xs:simpleType name="Job">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="Task1" />  
 <xs:enumeration value="Task2" />  
 <xs:enumeration value="Task3" />  
 </xs:restriction>  
 </xs:simpleType>  
   
 <xs:element name="Tasking">  
 <xs:complexType>  
 <xs:sequence minOccurs="0" maxOccurs="unbounded">  
 <xs:element name="Staffing">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="Employee" type="Name" />  
 <xs:element name="Task" type="Job" />  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
   
</xs:schema>

Successor XML Schema

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  
   
 <xs:simpleType name="ID">  
 <xs:restriction base="xs:string">  
 <xs:enumeration value="Node1" />  
 <xs:enumeration value="Node2" />  
 <xs:enumeration value="Node3" />  
 </xs:restriction>  
 </xs:simpleType>  
   
 <xs:element name="Network">  
 <xs:complexType>  
 <xs:sequence minOccurs="0" maxOccurs="unbounded">  
 <xs:element name="Successor">  
 <xs:complexType>  
 <xs:sequence>  
 <xs:element name="Node1" type="ID" />  
 <xs:element name="Node2" type="ID" />  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
 </xs:sequence>  
 </xs:complexType>  
 </xs:element>  
   
</xs:schema>