Cycles in SML Models

Working Document

Virginia Smith

1. Terminology

SML reference — An SML reference is a link from one element to another element within the same model (but not necessarily in the same document). An element information item in an SML instance document is an SML reference if and only if it has an attribute information item whose {name} is "ref" and whose {namespace} is <sml namespace> and whose {normalized value}, after whitespace normalization using "collapse" following schema rules, is either "true" or "1". (This is Sandy's definition from http://lists.w3.org/Archives/Public/public-sml/2007Aug/att-0086/SML_References.html.

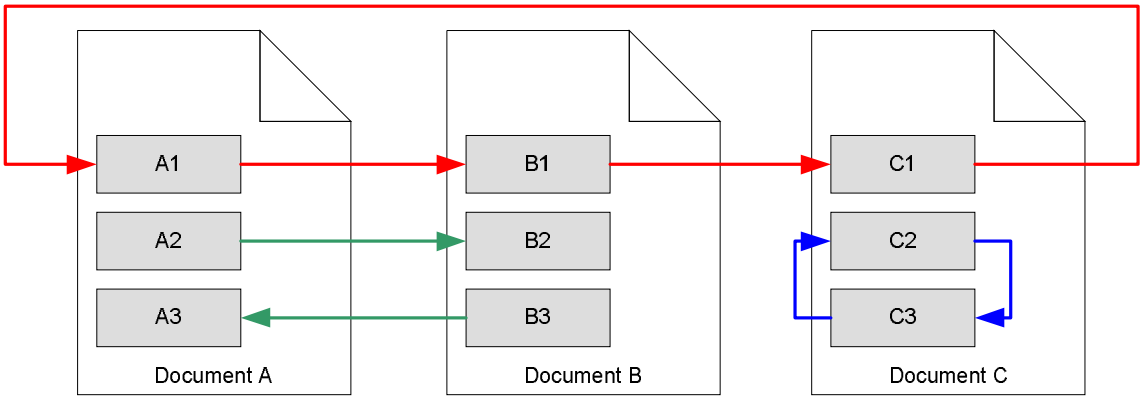
Reference source	The reference source is the element that contains the attribute "sml:ref" set to "true" or "1". (From SML 1.1, section 3.1)
Reference target	The reference target is the element referenced within the reference source using a reference scheme such as sml:uri. An SML reference can be an inter-document reference or an intra-document reference.
SML inter-document reference	An SML reference that is resolved to an information item from another document in the same SML model. (SML 1.1 actually states that the information item must be an element – section 3.1.2.4)
SML intra-document reference	An SML reference that is resolved to an information item (or element) from the same document that contains the SML reference.
Document-based cycle	Consider a directed graph whose nodes are the instance documents in an SML model that contain either an SML reference or the target information item for an SML reference and whose edges are instances of the SML reference. The edge (arc) is directed from the document containing the reference to the document containing the target. A cycle results when a path can be traced along the edges that encounters a document node already encountered previously in the same path.
Element-based cycle	A cycle is formed for an element E if the path that is formed by recursively following an SML reference that is a descendant of E to its target leads back to E.

2. Element- vs. Document-Based Cycles

In the current SML 1.1 draft specification, there is a mismatch between how SML references are defined and how cycles are defined. References are defined as a 'connection' between elements and cycles are defined based on 'connections' between documents. This section discusses several problems that arise from this mismatch.

The following diagram shows several scenarios for SML references and their targets. The table indicates whether each cycle in the diagram would be considered a valid cycle in an element-based graph or a document-based graph.

Figure 1 General cycle scenarios



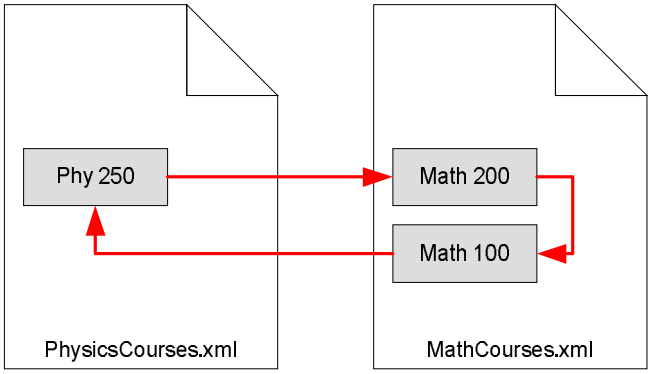
Cycle	Document-based cycle?	Element-based cycle?
Red Cycle (valid cycle)	ü	ü
Green Cycle (unrelated reference cycle)	ü (false positive)	û
Blue Cycle (intra-document cycle)	û (false negative)	ü

2.1. Valid Cycles

The red cycle in Figure 1 illustrates the most obvious example of a cycle. This would be considered a cycle, whether its representative directed graph is element-based or document-based.

The diagram below shows an example of a valid cycle where the references refer to prerequisite courses. This example depicts an undesirable condition where a student cannot register for one of these courses because it is impossible to meet the prerequisite requirement.

Figure 2 Valid cycle example



The following SML fragments illustrate the use of reference nodes of type "PrerequisiteType" to create a cycle as illustrated in the above diagram:

```
<Course>
  <Name>Math200</Name>
  <Prerequisite sml:ref="true">
    <sml:uri>
    http://www.university.example.org/Universities/MIT/MathCourses.xml#xmlns(u=http://www
    .university.example.org/ns) (/u:Courses/u:Course[u:Name='Math100'])
    </sml:uri>
  </Prerequisite>
</Course>
 . . .
<Course>
  <Name>Math100</Name>
  <Prerequisite sml:ref="true">
    <sml:uri>
    http://www.university.example.org/Universities/MIT/PhysicsCourses.xml#xmlns(u=http://
    www.university.example.org/ns) (/u:Courses/u:Course[u:Name='Phy250'])
    </sml:uri>
  </Prerequisite>
```

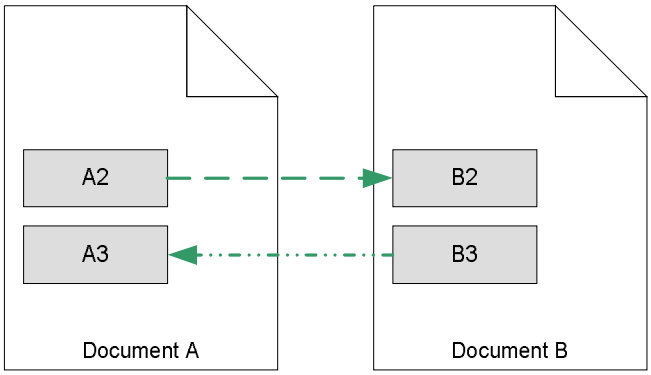
```
</Course>
 . . .
<Course>
  <Name>Phy250</Name>
  <Prerequisite sml:ref="true">
    <sml:uri>
    http://www.university.example.org/Universities/MIT/MathCourses.xml#xmlns(u=http://www
    .university.example.org/ns) (/u:Courses/u:Course[u:Name='Math200'])
    </sml:uri>
  </Prerequisite>
</Course>
```

2.2. Unrelated References

The green cycle in Figure 1 illustrates the unrelated reference problem. Here the reference from A2 to B2 and the reference from B3 to A3 are completely unrelated. This is not a valid cycle (of element references) but will appear as a cycle in a document-based graph.

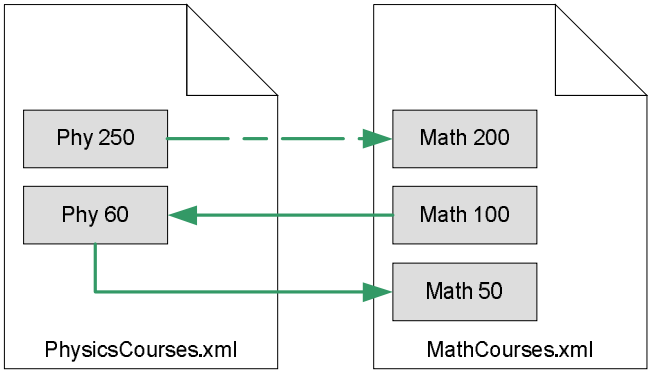
This false positive can be eliminated if the references are "typed", as in the following diagram, based on the elements (or element types) that they link. The graph node representing Document A will have 2 different edges emanating from it. Since the edges are of different types, no cycle is detected.

Figure 3 Unrelated reference cycle



The following diagram illustrates a condition where Math and Physics courses exchange prerequisites but, in fact, no cycle is created. This requires that the edges representing the links from the source to the target are somehow identified based on the elements linked rather than the document. In other words, the edges would be typed based on the elements linked.

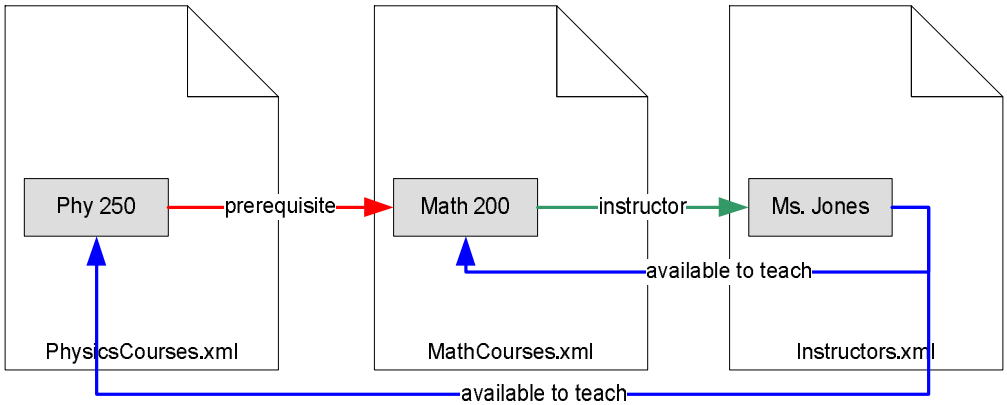
Figure 4 Unrelated reference cycle example #1



The question remains: how will we differentiate these links? In the example above, using the element's type (e.g., ClassType) will not be sufficient.

And, in case you don't have a headache yet, consider the following example of unrelated references that may look like a cycle but do not create a valid cycle. It is crucial that the "prerequisite", "instructor", and "available to teach" references be identified as separate edge types.

Figure 5 Unrelated references example #2

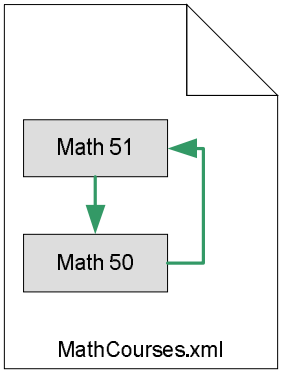


2.3. Intra-document References

The blue cycle in Figure 1 illustrates the intra-document reference problem. Here the reference is from C2 to C3 and from C3 back to C2. The elements C2 and C3 are contained in the same document. In this case, a cycle exists between references completely within a single document and does not appear as a cycle in a document-based graph.

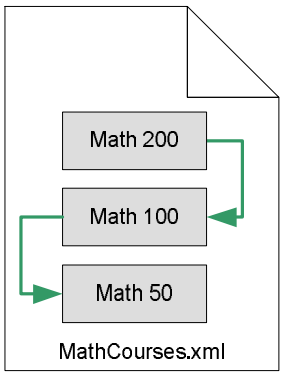
This scenario is illustrated in the following diagram where 2 Math courses have each other assigned as prerequisites. This anomaly will not be detected in a document-based graph where the edges only represent links between documents. This is a valid cycle and will be detected only if there is element-based cycle detection inside documents.

Figure 6 Intra-document references creating a valid cycle



The above scenario could be represented in a document-based graph by having an edge that points from the document node back to itself. However, we would still need to differentiate the above valid intra-document cycle from the scenario below where the edges would point back to the document node itself but is not actually a valid cycle. We cannot handle these scenarios correctly unless we use an element-based graph.

Figure 7 Intra-document references with no cycle



2.4. Requirements

2.4.1. Ability to detect SML reference cycles

Cycles must be detectable so that untenable situations such as described in section 2.1 can be flagged as error conditions.

?? what is a good scenario for allowing cycles?

2.5. Constraints

2.5.1. Reasonable implementation effort

If the implementation of an SML validator is not reasonably achievable, the adoption of SML will be negatively impacted.

Two areas in the implementation are potential problems:

1. Detecting cycles in general. Based on comments from HP and IBM implementation engineers, we do not believe this presents a challenge to the implementer.
2. Detecting cycles in a persistent data store. The concern was raised by Microsoft that their specific implementation of storing SML models in a database could make it difficult to detect cycles defined at the element level. One of the members of the WG (Kumar) is researching this.

2.6. Analysis

An element-based graph appears to be necessary to handle scenarios where the graph path includes intra-document references.

A document-based graph where the edges are "typed" by the reference source elements would be sufficient to identify valid cycles of inter-document references and correctly identify when unrelated references do not create a valid cycle. However, the question of how we type this edge still remains. This "edge typing" requires some knowledge of the reference elements themselves and the ability to distinguish between them. We may not be able to do that based on the complex type of the element. In Figure 4 above, there is no real cycle but the types of the elements will probably be the same so identifying the edge simply based on a complex type is not sufficient.

We may have to place a requirement on the SML model creator to create references between elements with careful consideration (modeler beware!). If cycles are based on the type of the reference which creates the edges in the graph, care must be taken to create the node types and use them in an unambiguous manner. At a minimum, if the reference nodes differ semantically they should differ in type.

2.7. Conclusion

While an SML validator can, in many cases, detect cycles using document-based graphs, knowledge of the reference elements themselves will be necessary to avoid false positives (as in the unrelated reference example) and to correctly handle intra-document references and cycles.

With this in mind, SML should define cycles as element-based. This would not preclude implementers from using document-based cycles (with typed edges) as an optimization when possible.

We could also consider placing explicit restrictions (or at least guidelines) on the use of SML references, such requiring a complex type definition for each semantically unique reference type and/or no intra-document references.

3. Proposals

Assuming that we define cycles based on elements, the next step is to determine the following:

- Where should the acyclic restriction be placed, on the SML reference or on its parent? For example, on the Node element or on it's Left/Right reference elements)
- What are the conditions for selecting the next arc or node when constructing a directed graph for the purposes of detecting cycles?

The following sections provide a new definition of a cycle and detail the proposals made to date.

3.1. Cycle Definition

Cycles will be defined in SML based on elements. Update the SML specification, section 4.3.1, with this new definition.

```
A cycle is formed for an element E if the path that is formed by
recursively following an SML reference that is a descendant of E to its
target leads back to E.
```

3.2. Syntax/Semantics

3.2.1. Proposal 1 – Reference Ancestor Element Annotation

This proposal puts the acyclic restriction in the <annotation> of a complexType of the SML reference parent (or ancestor) rather than as an attribute of the SML reference. Multiple sml:acyclic elements can be specified in the case where the type content contains multiple (different) SML references.

```
Model validators that conform to this specification MUST support the
sml:acyclic element on any <xs:complexType> element in a schema
document as shown below.

<annotation>
    <appinfo>
        <sml:acyclic ref="restricted xpath expression"/>
    </appinfo>
</annotation>
```

Example schema fragment:

```
<xs:complexType name="PrerequisiteRefType">
    <xs:sequence>
        <xs:any/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="CourseType">
  <xs:annotation>
    <xs:appinfo>
      <sml:acyclic ref="./Prerequisite"/>
    </xs:appinfo>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Name"/>
    <xs:element name="Prerequisite" sml:ref="true" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="Course" type="CourseType"/>
```

Example instance fragment:

```
<Course>
    <Name>Math101</Name>
    <Prerequisite sml:ref="true">
        <sml:uri>URI1</sml:uri>
    </Prerequisite>
    <Prerequisite sml:ref="true">
        <sml:uri>URI2</sml:uri>
    </Prerequisite>
</Course>
```

Notes

- In the example above, if the multiple Prerequisite elements were placed in a contain "Prerequisites", the annotation would still be placed on the CourseType complexType. The xpath expression in the sml:acyclic element would need to be adjusted accordingly, e.g., to ".//Prerequisite".
- A complexType could also contain multiple references of the same name/type, e.g., multiple prerequisite courses.

- The graph nodes consist of any element that is an instance of a complexType that contains an sml:acyclic annotation and its derived types. The SML references specified in the sml:acyclic element define the graph arcs.
- If an element contains an SML reference that is not mentioned in an sml:acyclic annotation, this SML reference is allowed to be cyclic.

3.2.2. Proposal 2 –Reference Ancestor Element Annotation with Named Cycles

This is the same as Proposal 1 except that the sml:acyclic has a "name" attribute allowing cycles to be defined that incorporate different target node types and different reference types using the cycle "name".

```
Model validators that conform to this specification MUST support the
sml:acyclic element on any <xs:complexType> element in a schema
document as shown below.

<annotation>
  <appinfo>
    <sml:acyclic name="<string>" ref="<restricted xpath expression>"/>
  </appinfo>
</annotation>
```

Example schema fragment:

```
<xs:complexType name="PersonType">
    <xs:annotation>
        <xs:appinfo>
            <sml:acyclic name="x1" ref="./Child"/>
        </xs:appinfo>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="Name"/>
        <xs:element name="Child" sml:ref="true"/>
        <xs:element name="Friend" sml:ref="true"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="Person" type="PersonType"></xs:element>
```

Example instance fragment:

```
<Person>
    <Name>Mary</Name>
    <Child sml:ref="true">
        <sml:uri>URI</sml:uri>
    </Child>
    <Friend sml:ref="true">
        <sml:uri>URI</sml:uri>
    </Friend>
</Module>
```

Notes

- The schema author must basically copy/paste this acyclic information to all complexTypes that participate in this cycle. An example of this is a dependency relationships among deployment CIs of different types.
- Cycles could be defined using different links (SML reference elements) within the cycle since cycles are created by following the cycle "name". Not only can different complexTypes participate in the

cycle but also different SML references (relationships/links) since the next arc is found by looking for the cycle name. This seems unnecessarily complicated and I can't think of a reasonable use case for this.

3.2.3. Proposal 3 – SML Reference Attribute with Xpath Expression

This proposal places the sml:acyclic constraint on the SML reference itself rather than on it parent or ancestor. This means that a relationship (reference) can be defined independent of its use in the model.

```
Model validators that conform to this specification MUST support the
sml:acyclic attribute on any complexType as shown below.

<xs:attribute name="sml:acyclic" type="<restricted xpath expression>"/>
```

Example schema fragment:

```
<xs:complexType name="dependencyRefType">
    <xs:sequence>
        <xs:any/>
    </xs:sequence>
    <xs:attribute name="sml:acyclic" use="required"/>
</xs:complexType>
<xs:complexType name="otherRefType">
    <xs:sequence>
        <xs:any/>
    </xs:sequence>
    <xs:attribute name="sml:acyclic" use="required"/>
</xs:complexType>
<xs:complexType name="nodeType">
    <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
        <xs:element name="DependsOn" type="dependencyRefType"/>
        <xs:element name="OtherRef" type="otherRefType"/>
</xs:complexType>
<xs:element name="Module" type="nodeType"></xs:element>
```

Example instance fragment:

```
<Module>
    <Name>App1</Name>
    <DependsOn sml:ref="true" sml:acyclic="./DependsOn">
        <sml:uri>http://example.com#AnyElement</sml:uri>
    </DependsOn>
    <OtherRef sml:ref="true" sml:acyclic="./DifferentRef">
        <sml:uri>http://example.com#AnyElement</sml:uri>
    </OtherRef>
</Module>
```

Notes

- *Instance* document creator must know what the value of sml:acyclic should be (usually "./<same name>" unless a global element is defined and referenced.
- Cycles can be defined by incorporating different elements/types. The next node in the cycle is found by following an SML reference.
- Cycles can also be defined using different links (SML reference elements) since the next SML reference is discovered by evaluating the sml:acyclic xpath expression, which may not match the SML reference to which it is attached

- Prohibited cycles are defined "on the fly". That is, any element can be part of the cycle and any SML reference can be an arc in a cycle.

3.2.4. Proposal 4 – SML Reference Boolean Attribute

This proposal is the same as Proposal 3 except that only SML references of a single complexType or any derived types are considered as part of the same cycle. There is no xpath expression required to locate the next reference (arc). The sml:acyclic constraint is placed on the SML reference. A global SML reference can be defined independent of its use in the model.

This proposal is very close to the original proposal in the submission specification except that graph nodes are elements not documents.

```
Model validators that conform to this specification MUST support the
sml:acyclic attribute on any complexType as shown below.

<xs:attribute name="sml:acyclic" type="xs:boolean"/>
```

Example schema fragment:

```
<xs:complexType name="nodeRefType">
    <xs:sequence>
        <xs:any/>
    </xs:sequence>
    <xs:attribute name="sml:acyclic" default="true"/>
</xs:complexType>
<xs:complexType name="nodeType">
    <xs:sequence>
        <xs:element name="ID" type="xs:integer"/>
        <xs:element name="Value" type="xs:string"/>
        <xs:element name="Left" type="nodeRefType"/>
        <xs:element name="Right" type="nodeRefType"/>
     </xs:sequence>
</xs:complexType>
<xs:element name="Node" type="nodeType"/>
```

Example instance fragment:

```
<Node>
    <ID>1</ID>
    <Value>App1</Value>
    <Left sml:ref="true">
       <sml:uri>xpointer(//Node[ID='5']</sml:uri>
    </Left>
    <Right sml:ref="true">
        <sml:uri> xpointer(//Node[ID='6']</sml:uri>
    </Right>
</Node>
```

Notes

- The arcs in the graph consist of all SML references of the same type and derived types. The nodes in the graph consist of all elements that are *pointed to* by one of these SML references. The key here is that a directed graph is created by first determining the arcs (SML references) and only then can the nodes of the graph be discovered (the targets of the references).
- It might be hard to find that first node of the graph when SML references determine which nodes are part of the graph. For example, a Prerequisite SML reference could be a child of Course or a descendant of Course (a child of Prerequisites container, for example). However, we don't really need

to determine the 1st node (the one that starts a possible cycle) because if there is a cycle involving this node, then an SML reference will point to it and, therefore, it will be automatically added to the graph.

- To detect a cycle when an element contains 2 refs of the same type (e.g. a Node that has Left and Right references, both of type nodeRefType), one must iteratively follow both refs - first follow one ref and if no cycle is detected then back up and follow the second ref. This would be repeated every time a target node contained more than one reference of the same type. (These permutations could be large.) This proposal will detect a cycle such as: N1's Left ref points to N2, N2's Right ref points to N1. This cycle is detected since all references of the same type are considered in cycle detection.
- Consider the example using a Person element that has a ref to both Child (of type ChildRefType) and Friend (of type FriendRefType) and these 2 ref types are both derived from PersonRefType. If a PersonRefType has an attribute sml:acyclic=true, then cycle detection for Person element would involving following all Child and Friend links to detect a cycle. So, if e1 has a friend e2 and e2 has a child e3 and e3 has a friend e1, then this would be a cycle if the acyclic attribute is true for PersonRefType. However, no cycle is detected if acyclic is specified as true on either ChildRefType or FriendRefType or each separately. This gives the schema author flexibility in defining cycle constraints.

3.3. Proposal Discussion

3.3.1. Comparison

To make a recommendation, I considered 4 scenarios:

S1 – Courses that have multiple prerequisite courses (acyclic).

S2 – Persons that reference multiple other persons as children (acyclic) or friends (cyclic).

S3 – Tree nodes that have references to other nodes as left or right children. Here both left and right must be considered as identical arcs when detecting a cycle. For example, an undesirable cycle could be formed when Node1's Left reference points to Node2 and Node2's Right reference points to Node1.

S4 – IT deployments where configuration items of different types reference each other with defined relationships, such as DependsOn or HostedOn.

Negatives:

1. Proposal P1 and P2 both require annotations in the complexType definition in the ancestor of the SML reference. This brings addition complexity both in terms of annotations, which are not inherited in Schema, and determining the correct ancestor on which to place the acyclic constraint.
2. Proposal P2 and P3 are unnecessarily complicated while not providing enough extra value for that complication. I could not think of any reasonable, real scenarios that would need the extra flexibility in defining cycles as being composed of just about any possible element or reference (in one cycle). In addition, proposal P3 degenerates to proposal P4 in all reasonable use cases I came up with.
3. Proposal P1 and P2 define the acyclic constraint as an attribute of the reference's parent elements (nodes) in the cycle which seems counter-intuitive.
4. Proposal P1 depends too much on the graph node hierarchy and, therefore, would not handle scenario S4 easily.
5. Proposal P1, P2, and P3 either do not handle scenario S3 at all or require too much complexity.
6. Proposal P3 requires that the sml:acyclic content repeat the name of the element which feels like a hack. It allows for completely different reference to be "next in line" but I haven't found a good use case for that.

7. Proposal P4 requires that, when detecting a cycle, all arcs must be of the same compexType or a derived type. (This is only a negative if you can find use cases where this is a problem… which I haven't.)

Positives:

1. Proposals P2 and P3 allow a lot of flexibility in defining what element instances in the XML documents can participate in a cycle (although this comes at a cost due to the complexity).
2. Proposal P3 and P4 allow for defining a relationship (graph arc) independent of the elements that participate in these relationships. This means the relationship can be reused. It also means that different complexTypes can participate in a cycle.
3. Proposal P3 and P4 defines the relationship on the SML reference type rather than on the element containing the reference and so the acyclic constraint travels with the reference and is inherited.
4. Proposal P4 is the only one that will handle scenario S3 simply.

3.3.2. Other Notes

3.3.2.1. Inheritance

- In schema, annotations are not inherited.
- In SML, because we are defining new component properties (acyclic, target*, key/keyref/unique), we can define inheritance rules of these properties, which can be different from schema's rules about annotations.
- For acyclic, we don't need to inherit it, because it applies to instances of the current type and all types derived from it. (Implementations would need to be very smart to avoid checking the same constraint multiple times if we do inherit it.)

3.3.2.2. Schema Author Responsibilities

Since sml:ref applies to elements and is no longer controlled by the schema document, it is worth pointing out that there are still requirements on the schema author with regard to SML references and cycles. Specifically, the schema author should:

- Identify any model elements that could or should include content via an SML reference and specify that element as a complexType with an open content model (for both elements and attributes). An alternative is to define the complexType as specifically allowing for sml:ref and the chosen reference scheme(s) as content.
- Identify any possible cycles in the model that would be problematic (for example, prerequisite courses) and add the sml:acyclic element/attribute appropriately.

Any open content allowed in a complexType creates a potential for an element of that type to contain an SML reference. This should be OK since the required content for the element must still be present. This means the required content cannot be "referenced out" to another document on the whim of the model instance creator.

4. Recommendation

Based on all of the above, my recommendation is that we go with Proposal 4 which seems to be the most understandable, intuitively makes sense, is the least complex, and handles (intuitively and simply) all the scenarios considered.

The following are my recommendations for changing the specification:

- Replace section 4.3.1 with the following text.

  ```
  sml:acyclic is used to specify that a cycle is not allowed for an SML
  reference type.
  ```

Consider a directed graph whose arcs are SML references of a given complexType (and any derived types) and whose nodes are the set of elements pointed to by any of these SML references. A cycle is formed for an element E if the path that is formed by recursively following an SML reference (of a given complexType and any derived types) from E to its target leads back to E.

Model validators that conform to this specification MUST support the sml:acyclic attribute on any <xs:complexType> element in a schema document. This is a boolean attribute and its value can be either true or false.

- Replace section 4.3.1.3 with the following text.

  If T is a complex type definition with {acyclic} true, then instances of T MUST NOT create cycles in any model. More precisely, the directed graph whose arcs are SML references of a given complexType (and any derived types) and whose nodes are all the elements pointed to by this set of SML references, must be acyclic.

- Replace section 8.1.1 with the following text.

  8.1.1 sml:acyclic

  Used to specify that an SML reference does not create any cycles in a model.

  If this attribute is set to true for a complex type D, then instances of D (including any derived types of D) that are SML references cannot create any cycles in a model. In the following example, HostedOnRefType is a complex type declaration whose instances can not create a cycle:

  <xs:complexType name="HostedOnRefType" sml:acyclic="true">

  ...

  </xs:complexType>

  If the sml:acyclic attribute is not specified or set to false for a complex type declaration ration, then instances of this type that are SML references may create cycles in a model.

5. Issues?

- "sml:ref" is required in the instance document to indicate that that the element is a reference no matter what is defined in the schema. "sml:acyclic" is only required when the reference cannot have cycles and must appear in a complexType in a schema document. There is a mismatch between requiring sml:acyclic in the schema but sml:ref only in the instance document. Instance document authors must remember to add sml:ref to all instance SML references whereas the sml:acyclic attribute is defined in the schema.

6. Acknowledgements

Yuan Chen (HP), James Lynn, and Sandy Gao contributed to this document.

Appendix: Algorithms

This appendix is here just to capture the algorithms discussed previously for the proposals. I think they only serve to complicate the description of the proposal. Graphs are more easily described by specifying the vertices and edges and then common graph algorithms will handle the cycle detection. However, I want to store the algorithms here for historical purposes.

Proposal 1

Replace section 4.3.1.3 in the SML specification with the following text.

> For any element information item E of type P that contains an sml:acyclic reference in its annotation, let R be the node that is referenced by the ref attribute of the sml:acyclic element in the annotation for E.
>
> 1. If R doesn't exist or R is not a reference element or R is a null reference, stop. No cycle is detected.
> 2. Otherwise deref() R to its target T.
>
> 2.1 If T is absent, OK.
>
> 2.2 If T is the same as E, then a cycle is detected and this is an error.
>
> 2.3 If T is of type P or a type derived from it, then let R the node that is referenced by the ref attribute of the sml:acyclic element in the annotation for T and repeat from step 1.
>
> 2.4 Otherwise, stop. No cycle is detected. (T is of a different type than E.)

Proposal 2

Replace section 4.3.1.3 in the SML specification with the following text.

> For any element information item E of type P or a type derived from P (e.g. "Person"), let R be the node that is referenced by the xpath expression (*evaluated in the context of E*) in the "ref" attribute of the sml:acyclic element in the annotation for E. Let C be the cycle name that is specified in the name attribute of same sml:acyclic element (e.g. "x1").
>
> 1. If R doesn't exist or R is not a reference element or R is a null reference, stop. No cycle is detected.
> 2. Otherwise deref() R to its target T.
>
> 2.1 If T is absent, OK.
>
> 2.2 If T is the same as E, then a cycle is detected and this is an error.
>
> 2.3 Otherwise, let R be the node that is referenced by the ref attribute of an sml:acyclic element in the annotation for T whose name attribute is equal to C. Then repeat from step 1.

Proposal 3

Replace section 4.3.1.3 in the SML specification with the following text.

For any element information item E (e.g. "Module"), let R be an SML reference (e.g. "DependsOn") that is a child of E and contains an sml:acyclic attribute.

1. If R doesn't exist or R is not a reference element or R is a null reference, stop. No cycle is detected.

2. Otherwise deref() R to its target T.

2.1 If T is absent, OK.

2.2 If T is the same as E, then a cycle is detected and this is an error.

2.3 Otherwise, let R be the node that is referenced by the xpath expression (*evaluated in the context of T*) in the sml:acyclic attribute of R and repeat from step 1.

Proposal 4

Replace section 4.3.1.3 in the SML specification with the following text.

For any element information item E, let R be an SML reference that is a child of E. Let RT be the type of R.

1. If R doesn't exist or R is a null reference, stop. No cycle is detected.

2. Otherwise deref() R to its target element T.

2.1 If T is absent, OK.

2.2 If T is the same as E, then a cycle is detected and this is an error.

2.3 Otherwise, let R be **any SML reference of type RT** that is contained in T. If there is more than one SML reference of type RT contained in T, then, select the "next" one and repeat from step 1. If no cycle is confirmed by following this chosen SML reference, return and select the next SML reference of type RT and continue with step 1 as before.