## Modeling Software

Recently I was tasked to write software to convert XML Metadata Interchange (XMI) files to Graphic Markup Language (GraphML) files:



I wrote the software. Afterwards, in my spare time, I created an Alloy model. Ideally, I would have created the model first and then written the software, but, alas, I was not being paid to develop a model. My model revealed some interesting things:

1. My software has a data leak. Someone using my software could gain access to sensitive information in the XMI file (e.g., author’s name, author’s email address).
2. My software has a bug. The Alloy Analyzer found the bug during the course of running 2 billion tests on my model.

Below is a description of the task, the model that I created, and how I used the Alloy Analyzer to uncover the data leak and the bug.

An XMI file is an XML encoding of a UML model. Here is a portion of an XMI file:

<xmi:XMI xmlns:uml='http://www.omg.org/spec/UML/20131001'
 xmlns:xmi='http://www.omg.org/spec/XMI/20131001'
 xmlns:StandardProfile='http://www.omg.org/spec/UML/20131001/StandardProfile'
 xmlns:Validation\_Profile='http://www.magicdraw.com/schemas/Validation\_Profile.xmi'
 xmlns:MagicDraw\_Profile='http://www.omg.org/spec/UML/20131001/MagicDrawProfile'
 xmlns:MD\_Customization\_for\_Requirements\_\_additional\_stereotypes=…'
 xmlns:sysml='http://www.omg.org/spec/SysML/20150709/SysML'
 xmlns:UserDefinedSterotypes='http://www.magicdraw.com/schemas/UserDefinedSterotypes.xmi'
 xmlns:DSL\_Customization='http://www.magicdraw.com/schemas/DSL\_Customization.xmi'
 xmlns:MD\_Customization\_for\_SysML\_\_additional\_stereotypes=…'>
 <xmi:Documentation>
 <xmi:exporter>MagicDraw UML</xmi:exporter>
 <xmi:exporterVersion>18.5</xmi:exporterVersion>
 </xmi:Documentation>
 <xmi:Extension extender='MagicDraw UML 18.5'>
 <plugin pluginName='SysML' pluginVersion='18.5'/>
 <plugin pluginName='Cameo Requirements Modeler' pluginVersion='18.5'/>
 <req\_resource resourceID='1480' resourceName='Cameo Requirements Modeler'
 resourceValueName='Requirement Diagram'/>
 <req\_resource resourceID='1440' resourceName='SysML'
 resourceValueName='SysML Activity Diagram'/>
 …
 </xmi:Extension>
 <uml:Model xmi:type='uml:Model' xmi:id='eee\_1045467100313\_135436\_1' name='Model'>
 <ownedComment xmi:type='uml:Comment'
 xmi:id='\_18\_1beta\_8c90288\_1415101994820\_370998\_4336'
 body=**'Author:JohnDoe**.&#10;Created:7/17/17 8:23
 AM.&#10;Title:.&#10;Comment:.&#10;'>
 <annotatedElement xmi:idref='eee\_1045467100313\_135436\_1'/>
 </ownedComment>
 <packagedElement xmi:type='uml:Package'
 xmi:id='\_18\_5\_1\_5a901aa\_1500294237329\_995324\_4667'
 name='SystemModel'>
 <xmi:Extension extender='MagicDraw UML 18.5'>
 <modelExtension>
 <ownedDiagram xmi:type='uml:Diagram' …

GraphML is a standard way of representing graphs. A GraphML file consists of nodes and edges. Here is rendering of a GraphML file that contains 11 nodes and 12 edges:



Here is the GraphML file:

<graphml>
 <graph id="G" edgedefault="undirected">
 <node id="n0"/>
 <node id="n1"/>
 <node id="n2"/>
 <node id="n3"/>
 <node id="n4"/>
 <node id="n5"/>
 <node id="n6"/>
 <node id="n7"/>
 <node id="n8"/>
 <node id="n9"/>
 <node id="n10"/>
 <edge source="n0" target="n2"/>
 <edge source="n1" target="n2"/>
 <edge source="n2" target="n3"/>
 <edge source="n3" target="n5"/>
 <edge source="n3" target="n4"/>
 <edge source="n4" target="n6"/>
 <edge source="n6" target="n5"/>
 <edge source="n5" target="n7"/>
 <edge source="n6" target="n8"/>
 <edge source="n8" target="n7"/>
 <edge source="n8" target="n9"/>
 <edge source="n8" target="n10"/>
 </graph>
</graphml>

As you know, when writing software you must provide a step-by-step set of instructions to the computer. A model of the software, however, doesn’t do that. A model merely specifies the input structures, the output structures, and the constraints on the input and output structures. The Alloy Analyzer determines if instances can be generated which satisfy the constraints.

There are four key points to remember when modeling software:

1. **Don’t model everything**. Pick out a tricky or critical part of the problem and just model that.

2. **Abstract the problem**. XMI and GraphML files contain lots of details: there are XML namespaces, some data is expressed using XML attributes while other data is expressed using XML elements, and in XMI there are elements for visualization. Such details are not relevant. The model just needs a few high-level concepts such as Element, Node, and Edge.

3. **Don’t give step-by-step instructions on how to solve the problem**. Just state the structure of the XMI file, the structure of the GraphML file, and the constraints on the structures. The Alloy Analyzer will determine if the constraints can be satisfied—if a GraphML file can be generated from an XMI file.

4. **Why are you creating a model**? What questions do you want the model to answer? Here are some possibilities: You want to model the software to gain a deeper understanding of the problem. You want to know if the software has data leaks. You want to know if the software has bugs.

Okay! Let’s model.

There are two structures to model: the input XMI file and the output GraphML file.

## Components of XMI

XMI files contain elements and each element has a name:

**sig** Element {
 name: Name
}

Each element may have child elements.

**sig** Element {
 name: Name,
 children: **set** Element
}

Some elements have an aggregation attribute.

**sig** Element {
 name: Name,
 aggregation: **lone** Aggregation,
 children: **set** Element
}

XMI is XML and XML always has a root element:

**one** **sig** Root **extends** Element {}

Create a set of names:

**sig** Name {}

One of the elements has the name SystemsContext:

**one** **sig** SystemsContext **extends** Name {}

Create a set of aggregation values:

**sig** Aggregation {}

Here is a simple XMI file, displayed as a tree:



Okay, that’s it for modeling the structure of the input XMI file (tree). Now let’s define the constraints on trees in general and on XMI trees in particular.

## Constraints on trees

The above Element signature defines a set of Element. We want every member of that set to be contained within the Root element:

**fact** No\_disconnected\_elements {
 **all** e: Element |
 (e = Root) **or** (e **in** Root.^children)
}

An element cannot be a descendent of itself (i.e., no cycles in trees).

**fact** No\_loops {
 **no** e: Element | e **in** e.^children
}

Each element has one parent.

**fact** Each\_element\_has\_one\_parent {
 **no** **disj** e, e', e'': Element |
 (e **in** e'.children) **and** (e **in** e''.children)
}

## Constraints on XMI trees

There must be exactly one element with the name 'SystemsContext'

**fact** One\_element\_named\_SystemsContext {
 **one** e: Element | (e.name = SystemsContext)
}

## Components of GraphML

A GraphML file contains nodes and edges. Each node has a name. Each edge references a source node and a target node.

**sig** Node {
 name: Name
}

**sig** Edge {
 source: Node,
 target: Node
}

**one** **sig** GraphML {
 nodes: **set** Node,
 edges: **set** Edge
}

To help with debugging, it will be useful to add a field to Node to reference the Element that is the source of the Node's data:

**sig** Node {
 name: Name,
 represents: Element
}

The GraphML contains nodes corresponding to only some of the XMI elements: the SystemsContext element and its descendent aggregation elements:



It will be useful to create a helper signature with a field that contains the SystemsContext element and its descendent aggregation elements (i.e., all the elements for which we need a node).

**one** **sig** Relevant {
 elements: **set** Element
} {
 elements = {e: Element | (e **in** SystemsContext.~(Element <: name).\*children) **and**
 ((e.name = SystemsContext) **or** (e.aggregation != none))}
}

## Constraints on GraphML

The GraphML must contain a node for the SystemsContext element and a node for each aggregation element that descends from the SystemsContext element.

**fact** GraphML\_nodes {
 #GraphML.nodes = #Relevant.elements
 **all** e: Relevant.elements |
 **one** n: Node |
 (n.name = e.name) **and** (n **in** GraphML.nodes)
}

The GraphML must contain an edge for each aggregation element. The source of the edge is the node corresponding to the aggregation element and the target of the edge is the node corresponding to the SystemsContext element.

**fact** GraphML\_edges {
 #GraphML.edges = minus[#Relevant.elements,1]
 **let** targetNode = SystemsContext.~(Node <: name) |
 **all** sourceNode: Node - targetNode |
 **some** e: Edge |
 (e.target = targetNode) **and** (e.source = sourceNode) **and** (e **in** GraphML.edges)
}

I ran the Alloy Analyzer on the model. It generated many instances. Below is one of the XMI instances it generated:



The Analyzer generated a node for the relevant elements: one node for the SystemsContext element and two nodes for each of the two descendent aggregation elements:



For each aggregation node, it created an edge to the SystemsContext node:



**Recap**: The model merely specifies the structure of the XMI, the structure of the GraphML, and the constraints on the XMI and GraphML. The Alloy Analyzer generates XMI instances and the corresponding GraphML. In other words, we didn’t have to provide to the computer a step-by-step set of instructions on how to generate GraphML from XMI, the Alloy Analyzer generated the GraphML just by analyzing the constraints on the XMI and the GraphML.

Is the model correct? Let’s write some asserts and have the Alloy Analyzer check the model. This is where the real power of Alloy is.

Through accidental oversight, the represents field was not removed. Consequently, a person with access to the GraphML also has access to the XMI tree and therefore can see more data (e.g., Author data, email data) than the person should. In other words, there is data leakage.



The value of each Node must be just a Name. Thus, the values of the fields within Node must be a subset of the set of Names.

**assert** No\_data\_leaks\_from\_Node {
 **all** f : Node$.subfields| f.value[Node] **in** Name
}

That assert uses the Alloy meta capability, which allows you to 'iterate' over the fields of a signature. See <http://alloytools.org/quickguide/meta.html>

The assert produces counterexamples because each Node references an Element in the XMI tree, which gives access to all the (sensitive) data in the XMI.

We can perform other checks on the model (and thus, indirectly, check the software):

**Assert**: There is a node for every aggregation element.

**assert** There\_is\_a\_node\_for\_every\_aggregation\_element {
 **all** e: aggregation.Aggregation |
 **some** n: Node |
 (n.represents = e) **and** (n.name = e.name)
}

This should yield a counterexample because aggregation elements that do not descend from SystemsContext do not have a corresponding Node. Sure enough, Alloy found a counterexample.

**Assert**: There is exactly one node corresponding to the SystemsContext element.

**assert** There\_is\_one\_SystemsContext\_node {
 **one** e: Element |
 (e.name = SystemsContext) **and** (e in Root.\*children) **and**
 **one** n: Node |
 (n.name = SystemsContext) **and** (n.represents = e)
}

This should not yield a counterexample because we require a SystemsContext element and a node corresponding to it. Yikes! Alloy found a counterexample. Here’s the counterexample: The counterexample does indeed show one node within GraphML corresponding to the SystemsContext element, but it also shows a SystemsContext node outside GraphML. The model did not constrain the Node set to only nodes within GraphML. That is a bug. I fixed it by adding this constraint:

**fact** No\_disconnected\_nodes {
 **all** n: Node |
 n **in** GraphML.nodes
}

**Assert**: There is no node for an element that is not SystemsContext or an aggregation element.

**assert** There\_is\_no\_node\_for\_an\_element\_that\_is\_not\_SystemsContext\_or\_an\_aggregation\_element {
 **no** n: Node |
 (n.represents.name != SystemsContext) **and** (n.represents.aggregation = **none**)
}

This should not yield a counterexample because nodes must correspond to either the SystemsContext element or to an aggregation element. Sure enough, no counterexamples were found.