EPSILON DRIVER FOR ASTAH GSN

CHALLENGES IN IMPLEMENTING AN EPSILON DRIVER FOR ASTAH GSN

By

BARAN KAYA, M.Eng.

A Thesis
Submitted to the Department of Computing & Software
And the School of Graduate Studies
of McMaster University
in Partial Fulfillment of the Requirements
for the degree
Master of Engineering

McMaster University

Master of Engineering (2020) McMaster University

(Computing & Software) Hamilton, Ontario, Canada

TITLE: Challenges in Implementing an Epsilon Driver for Astah GSN

AUTHOR: Baran Kaya
M.Eng. (Software Engineering)
McMaster University, Hamilton, Canada

SUPERVISOR: Dr. Richard Paige

NUMBER OF PAGES:

ABSTRACT

Model-Driven Engineering (MDE) methods popularity on the rise especially in the Safety-Critical software systems development area. Goal Structuring Notation (GSN) diagrams are one of the most used models in this sector and it is becoming a standard for safety purposes. Epsilon is the Eclipse's Model-Driven platform for working on MDE models. It makes it easier to work on any kind of model with the right driver for it. Astah GSN is one of the most advanced software for working with GSN diagrams, however; it doesn't have any Epsilon drivers yet. In this project, a new Epsilon driver for Astah GSN models developed. While developing this driver, lots of were challenges encountered. This thesis report explains every challenge that was encountered while implementing the driver also testing methods that were used while evaluating the project results.

CONTENTS

1. INTRODUCTION

Intro

Explain sections!!!!

2. RELATED WORKS

2.1. Model-Driven Engineering

Model-Driven Engineering (MDE) is becoming the standard for some types of software development. Safety-critical systems development is one of the most used areas fro MDE methodologies. MDE makes it easier to design a system with a combination of domain-specific modelling languages (DSML) and transformation engines/generators [1]. DSMLs help developers to design better systems for their domains. Avionics, transportation systems and many others use DSML to create safer software systems. The second part which is transformation engines and generators helps programmers to transform models-to-models and generate source code or documents from the designed models.

MDE in Practice book

2.2. Safety Cases & Goal Structuring Notations

Structure of the safety section: a) What is safety? (Definitions etc); b) How is safety engineered? (process: STPA, STAMP, v-model); c) How is safety argued? (GSN, assurance cases)

2.3. Epsilon

Epsilon-EOL…

EMC layer-Epsilon book…

New driver-Bridging paper…

CHANGE!!!

Epsilon Astah-GSN driver project is my McMaster University Master of Engineering project. The main goal of the project is to use the Eclipse Epsilon Model-Driven Engineering tool to modify Astah GSN models. The integration between these two programs made with the Epsilon Model Connectivity Layer (EMC). This project is developed for General Motors.

Epsilon is an Eclipse plugin for Model-Driven Engineering processes. Epsilon website defines it as “Epsilon is a family of mature languages for automating common model-based software

engineering tasks, such as code generation, model-to-model transformation, and model validation, that work out of the box with EMF (including Xtext and Sirius), UML, Simulink, XML and other types of models." [1].

GSN stands for Goal Structuring Notation. GSN Working Group defines it as a visualization of safety argument elements and relations. Requirements, claims, evidences, and contexts are some of the safety argument elements [2]. GSN aims to show these elements and the connection/relationship between each element. Astah GSN is one of the modelling programs that specifically designed for creating and modifying GSN models.

3. REQUIREMENTS

The main goal of the project is to be able to use Epsilon on Astah GSN models so that users can work on the Astah GSN models. To do that, the driver to be able to parse the Astah GSN files correctly. Astah GSN saves its model files with '.agml' extension; however, it provides XMI import/export features for GSN models. Astah GSN encodes AGML files so users cannot reach these files content. All EOL functionality can be used within other Epsilon languages as well. Therefore, in this driver project XMI files used. Also, there was Plain XML driver for Epsilon, and it is used as a based version of the project driver.

Epsilon has one core language (Epsilon Object Language, EOL) and ten task-specific languages. Task-specific languages derived from EOL thus, integrating models to EOL is sufficient for all other language support. The first requirements show the EOL integration and the later ones show task-specific language integrations.

The requirements of this driver project don't collect from any users. Listed requirements generated by me after examining the GSN diagrams, Astah GSN XMI files, similar Epsilon drivers, and Epsilon source code. Element access requirements are coming from GSN standards and attribute access requirements are coming from both GSN and Astah GSN XMI file. For instance, ID access was originated from the GSN diagram's element ID on the other hand XMI:ID was originated from the XMI tag attribute which stores the unique ID value for each tag element.

Epsilon Astah GSN Requirements:

1. Users should be able to load Astah GSN models into the Epsilon in Eclipse IDE.

2. Users can read/access the GSN models with Epsilon Object Language (EOL).
   - 2.1. Users should be able to access the entire GSN model
   - 2.2. Users should be able to access certain types of elements
     - 2.2.1. Accessing to all nodes
     - 2.2.2. Accessing to all links
     - 2.2.3. Accessing certain types of nodes (e.g. goals)
     - 2.2.4. Accessing certain types of links (e.g. asserted evidence)
   - 2.3. Users should be able to access a specific element
     - 2.3.1. Accessing a specific node
     - 2.3.2. Accessing a specific link
   - 2.4. Users should be able to access elements' attribute values
     - 2.4.1. Accessing an element's content
     - 2.4.2. Accessing an element's ID
     - 2.4.3. Accessing an element's type

2.4.4. Accessing an element's XMI:ID
2.4.5. Access an element's XSI:TYPE
2.4.6. Access a link's target
2.4.7. Access a link's source

3. Users should be able to update the GSN models with EOL.
   3.1. Updating an element's content
   3.2. Updating an element's ID
   3.3. Updating an element's type
   3.4. Updating an element's XMI:ID
   3.5. Updating an element's XSI:TYPE
   3.6. Updating an element's target link
   3.7. Updating an element's source link

4. Users should be able to create new elements and append new elements into the GSN model with EOL.

5. Users should be able to delete elements in the GSN model with EOL.

6. Users should be able to validate the Astah GSN model with Epsilon Validation Language.

7. Users should be able to transform the Astah GSN model to another model with Epsilon Transformation Language.

8. Users should be able to generate code or text from the Astah GSN model with Epsilon Code Generation Language.

In this chapter, project requirements and how/where they are collected explained. The next chapter will explain the design decisions of the project and what kind of changes/updates made while developing the project.

4. DESIGN

Designing an Epsilon driver from scratch is very challenging. It also takes a lot of time but fortunately, Epsilon has lots of built-in and external drivers for similar models. The most similar drivers that could be used in the design of the Astah GSN driver were internal Plain-XML driver and external HTML driver. While designing the Astah GSN driver, these two drivers and their features helped a lot. Both plugin drivers and how they are used in this project will be explained in this section.

While developing the driver and the project, waterfall methodology used. But some changes are made while developing the project due to the uncertainties. The first step of the project was examining the GSN diagrams and learning the basics. The next step was looking out for similar Epsilon drivers, their element access operations and how they are parsing the model files. Then project requirements created based on collected information from GSN and similar drivers. The next step was looking for Epsilon driver plugin projects so that without I don't have to deal with Epsilon plugin operations. For that purpose, I used the Epsilon HTML driver and build Astah GSN parser on top of that project. The last step was testing the implementation of the driver. In this step, Epsilon ran in debug mode and each step investigated. Some necessary updates made in the source code and all of the changelog can be found in the project's GitHub repository.

Plain-XML driver comes with Epsilon installation. It is an internal model driver like EMF (Eclipse Modeling Framework) or UML (Unified Modeling Language) model drivers. Since Astah GSN uses XMI (XML Metadata Interchange) files, using the Plain-XML driver and its methods was made it easier to develop this project. At first, I examined the Plain-XML driver's source code and how it works. For that reason, I run Epsilon from source code in Eclipse and debugged the Plain-XML driver while running the EOL script on XML files.

Plain-XML driver can parse XML files and users could load, read and update XML models in Epsilon with this driver. However, this driver is not useful for Astah GSN XMI files. Because GSN XMI files store every element in the GSN model with the same tag name (*argumentElement*) but Plain-XML driver parses files via different tag names. User can access different elements, their child elements and their attributes with the tag name parameter. However, Astah GSN uses XML attributes to store every elements' values such as type, content, and ID. Therefore, the Plain-XML driver needed to be heavily modified for attribute values instead of tag names.

Unlike the Plain-XML driver, the HTML driver doesn't come with Epsilon installation. Instead, the user has to get the HTML driver from Epsilon Labs [4] GitHub page and then has to run Eclipse from source code to use the HTML driver itself. For the Astah GSN driver project, using the HTML driver as a base project made developing a driver plugin for Epsilon easier. Like the Plain-XML driver, I examined the HTML driver and its source code and debugged it several times. But, the HTML driver doesn't have as much code as the Plain-XML driver. It just has every necessary class link in the Plain-XML driver but most of its methods call Plain-XML driver's methods after parsing the HTML file. At the end of the day, HTML is a subset of the XML and using the Plain-XML

driver methods is for HTML driver's developers. However, that means I couldn't use the HTML driver source code as a base project because most of its class is useless for the Astah GSN XMI file parser. So instead, the HTML driver's source code used for plugin features. The names in the HTML driver's plugin packages changed to "Astah GSN" thus, new project packages for the project created. More details about HTML will be explained later in this section.

4.1. Implementing an Epsilon Driver for a Commercial Tool

There are several Goal Structuring Notation diagram tools out there. However, this project developed for General Motors and they were using Astah GSN for their main GSN diagram tool. Moreover, Astah GSN has the most features that other GSN tools don't have. But, this doesn't mean Astah GSN is perfect. It is a commercial tool and most of the methods that they used in Astah GSN are commercial secrets. That's why I couldn't use the Astah GSN model files (.agml) while developing the Epsilon driver. I have to use its XMI import/export function to access the GSN model.

XMI import/export functionality is a good feature but like Astah GSN it's not perfect. For instance, some node elements' in the XMI file uses the same type of attribute values and it's hard to identify elements' types. Also, the exported XMI file doesn't store the GSN diagram, it only stores elements. Thus after importing the XMI file back into Astah GSN, the user has to drag and drop each element from the right side to the main area to create the GSN diagram again. After dropping node elements, it connects them correctly via link elements but not storing the diagram is a downside of the Astah. Another con of the XMI file is not storing every link element. For example, it doesn't store Goal-to-Strategy link elements in the file and it only stores Strategy-to-Goal links in the Strategy element and not as an individual link element. This makes it harder to develop an Epsilon driver for Astah GSN.

4.2. XMI File and Element Attributes

XMI (XML Metamodel Interchange) is an OMG (Object Management Group) standard format for interchanging MOF (Meta Object Facility) models [5] such as GSN. Astah GSN could export its GSN models as XMI files. In XMI format, all elements use the same tag name except the root element. All element features like type, ID, even content stored in the element attributes. An XML parser can be able to parse the XMI files. The Plain-XML driver could parse the given Astah GSN XMI file but the user cannot access all types of elements because Plain-XML driver lacks parse by attribute features. The new Astah GSN driver provides attribute parser and other additional features

for Astah GSN models. With this driver, the user should be able to access or updated each elements' attributes via correct commands.

Goal Structuring Notation consists of six node types and two relationships (link) types. However, Astah GSN doesn't use the same element types as GSN Standards. Node elements are the same but Astah uses more than two link types in the XMI document. Table 1 shows the GSN Standard's element types and Table 2 shows the Astah GSN XMI file's element types.

Node Elements	Link Elements
• Goal • Strategy • Solution • Context • Assumption • Justification	• SupportedBy (goal-to-goal, goal-to-strategy, goal-to-solution, strategy-to-goal) • InContextOf (goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption and strategy-to-justification)

Table 1: GSN Model Standard Element Types

Node Elements	Link Elements
• Goal • Strategy • Solution • Context • Assumption • Justification	• Inference (Asserted Inference) • Evidence (Asserted Evidence) • Context (Asserted Context)

Table 2: Astah GSN Element Types

Astah GSN uses XML attributes to store all data in the XMI formatted file. All elements (except root) uses the same tag name but they all use different attributes and attribute values. Table 3 shows the GSN element's (tag name: argumentElement) attribute and their description.

Attribute Name	Element Type	Description
xsi:type	All elements	Element's type (Some node types have the same xsi:type attribute)
xmi:id	All elements	Unique ID for each element
ID	All elements	Element's ID shown in the GSN diagram (e.g. G1)
Description	All elements	Description of the element (Mostly empty)
Content	All elements	Content of the element that is shown on the GSN diagram
URL	Nodes (Solution, Context)	Hyperlink of the attached documents

Assumed	Nodes (Goal, Assumption, Justification)	Assumption element's value is true, Goal and Justification elements are false
ToBeSupported	Nodes (Goal, Assumption, Justification)	Undeveloped goal element's attribute value is true
DescribedInference	Nodes (Strategy)	Strategy element's InContextOf relationships with Goal elements
Target	Links	Link element's source node xmi:id (In Astah GSN, target and source are reversed)
Source	Links	Link element's target node xmi:id (In Astah GSN, target and source are reversed)

Table 3: Astah GSN XMI document attributes

4.3. Using the Epsilon HTML Driver

There are multiple built-in drivers for different models in the Epsilon. For instance, the Plain-XML driver is coming with Epsilon download. However, I needed to create a new driver plugin for Astah GSN models. Therefore, I searched for other model drivers for Epsilon that aren't built-in Epsilon. EpsilonLab GitHub page [3] has a few EMC drivers for several models such as HTML, JDBC, JSON, and more. The most similar model driver to Astah GSN was HTML due to the XML structure.

After cloning the EMC-HTML git repository, I tried to run it on the Epsilon source code. The first step was importing the 2 HTML project packages into Eclipse Epsilon workspace. Even though the HTML repository has 6 different packages, only 2 of them are necessary for running the HTML driver. The other 4 packages are test and example packages. After that step, the Epsilon source code rebuilds itself. Then running Epsilon on the new Eclipse application is sufficient for HTML driver. The new driver is an Epsilon model in the Run Configuration of every Epsilon languages. For testing purposes, I created a new EOL file. Then I created a new run configuration for this EOL file and HTML Document was one of the model options in the Model Selection tab. After selecting it, the EOL script will use an HTML file as a model.

Since I have no experience with Epsilon plugin development, I used an HTML driver as a base plugin project. I changed all names in the project and used a new image for the Astah GSN model selection tab. After that, I examine the HTML driver classes. The main class is called the HTML model and it invokes getter and setter classes for different functionalities. However, HTML getter and setter only call Plain-XML getter and setter functions.

I just used an HTML driver for plugin features and changed all models, getter, and setter classes/methods. HTML driver was using Java Jsoup library for HTML elements parse. This library could also parse the XML/XMI files but each time this library used, it adds <html> and other main tags into the XML file. So, this library changed to the Plain-XML driver's parser library.

4.4. Using the Epsilon Plain-XML Driver

HTML plugin was the most similar example to Astah GSN driver that I am going to work on. But parsing methods in HTML driver wasn't useful for the XMI file parser. That's why the HTML driver project used for only Epsilon plugin features and all other classes like the model, getter, and setter changed based on Plain-XML driver.

Plain-XML driver's main goal is parsing given XML files based on tag names. For this reason, it uses Java W3C Dom library's Node and Element classes. Each element represents a tag object. Elements store tag's attributes, text, and its child tags. Plain-XML driver's classes parse XML files based on tag names. However, the Astah GSN XMI file uses the same tag name for every element and it uses attributes to determine element types. Thus, the Plain-XML driver has to be modified for parsing attributes instead of tag names. To do these modifications model, type, getter, and setter classes and their methods have to change. The basic changes in each class will be explained.

The model class is the main class of the whole driver. It is responsible to file operations, model operations, invoking getter and setter classes, element creation, and removal operations. This class didn't modify much because file, model, getter, and setter operations are the same for XML and XMI files. The only modification made in the element creation function. Plain-XML driver's element creator function was using tag names for new elements. But, Astah GSN XMI driver doesn't require tag names because all elements use the same tag name. Thus this function changed for getting GSN element's type instead of getting the tag names as a function input.

Type class in Plain XML has 4 types for XML files. These are tag, attribute, reference, and child. But these types are not necessary for Astah GSN driver. Thus, types are changed to GSN model elements such as goal, strategy, solution, etc.

Getter and setter classes are heavily modified based on the Astah GSN XMI file. These two classes parse the XMI file based on attributes. All classes and their methods are explained in *Section 5: Implementation*.

4.5. Determining Element Types in the XMI File

Determining element types such as Goal, Solution or Asserted Evidence is one of the important parts of the Astah GSN Driver. In early design, each user queries were parsed before accessing the element in the XMI file. Thus, if the given query element wasn't in the GSN type, it would return an empty result. For example, a user can only get elements with IDs like G1, Sn4, C2, … These IDs gave by Astah GSN by element's type. Goal elements' ID starts with G, Solution with Sn and Context with C. However, element IDs don't have to start with element type letters. In this case, GSN type parse for custom IDs such as CA1, AR-C1 returned null and Epsilon returned error message for couldn't finding the element with given custom ID.

In later designs, custom ID access added to the driver. Because GSN standards don't require IDs to start with element type letters. That's why instead of returning null for custom ID queries, the driver compares every ID in the model and if it couldn't find it then it returns null. For custom IDs, GSN type parser still returns null but after that, it checks every element for custom ID probability.

Another update for not determining types with element ID would be the ".gsntype" query. Before this change, the node elements' type was found by element IDs and link elements' IDs were found by *xsi:type* attribute. After discovering custom IDs, the *gsntype* function has to change. However, there is a problem with determining element type without IDs. The only way to finding element types is the *xsi:type* attribute. But, some of the node elements use the same xsi:type attributes. For instance, Goal, Assumption, and Justification elements all use "ARM:Claim" value for *xsi:type* attribute. So, for determining types, I have to use other attributes. The only difference between Goal-Justification pair and Assumption elements is assumed attribute. For Assumption elements this attribute's value is true but for Goal and Justification elements this attribute's value is false. Now we can determine Assumption elements from Goal and Justification elements. For determining between Goal and Justification elements, there aren't any attributes. The only difference between these 2 elements is their connections. Goal elements can connect all 3 types of link elements but Justification elements can only connect to the Asserted Context element's target side. Thus, if the given element's *xmi:id* stored in one if the Asserted Context attributes' target attribute, that means the element's type is Justification.

The Goal-Justification situation is the same for Solution-Context pairs. Instead of "ARM:Claim" *xsi:type* attribute Solution-Context elements use "ARM:InformationElement". Similarly, Context elements can only be connected to Asserted Context links' target side. Hence, the same function used for determining Justification and Context elements. All 5 element examples could be seen below.

Goal element
<argumentElement xsi:type="ARM:Claim" xmi:id="_fvLpEJq4EeqyzooT9RpXrQ" id="G1" description="" content="Control System is acceptably safe to operate" assumed="false" toBeSupported="false"/>
Assumption element
<argumentElement xsi:type="ARM:Claim" xmi:id="_fvLpI5q4EeqyzooT9RpXrQ" id="A1" description="" content="All hazards have been identified" assumed="true" toBeSupported="false"/>
Justification element
<argumentElement xsi:type="ARM:Claim" xmi:id="_fvLpJJq4EeqyzooT9RpXrQ" id="J1" description="" content="SIL apportionment is correct and complete" assumed="false" toBeSupported="false"/>
Solution element
<argumentElement xsi:type="ARM:InformationElement" xmi:id="_fvLpE5q4EeqyzooT9RpXrQ" id="C1" description="" content="Operating Role and Context" url=""/>
Context element
<argumentElement xsi:type="ARM:InformationElement" xmi:id="_fvLpH5q4EeqyzooT9RpXrQ" id="Sn1" description="" content="Formal Verification" url=""/>
Strategy element
<argumentElement xsi:type="ARM:ArgumentReasoning" xmi:id="_fvLpF5q4EeqyzooT9RpXrQ" id="S1" description="" content="Argument over each identified hazards" describedInference="_fvLpM5q4EeqyzooT9RpXrQ _fvLpNJq4EeqyzooT9RpXrQ _fvLpNZq4EeqyzooT9RpXrQ"/>
Asserted Context element
<argumentElement xsi:type="ARM:AssertedContext" xmi:id="_fvLpJZq4EeqyzooT9RpXrQ" id="" description="" content="" source="_fvLpFJq4EeqyzooT9RpXrQ" target="_fvLpEJq4EeqyzooT9RpXrQ"/>
Asserted Inference element
<argumentElement xsi:type="ARM:AssertedInference" xmi:id="_fvLpJ5q4EeqyzooT9RpXrQ" id="" description="" content="" source="_fvLpEZq4EeqyzooT9RpXrQ" target="_fvLpEJq4EeqyzooT9RpXrQ"/>
Asserted Evidence element
<argumentElement xsi:type="ARM:AssertedEvidence" xmi:id="_fvLpL5q4EeqyzooT9RpXrQ" id="" description="" content="" source="_fvLpIZq4EeqyzooT9RpXrQ" target="_fvLpHZq4EeqyzooT9RpXrQ"/>

Table 4: Example Element Tags from Astah GSN XMI File

4.6. Problems with Astah GSN XMI Files

Astah GSN is one of the best GSN diagram tools right now but it isn't flawless. Its XMI import/export feature is not very good compared to its GSN diagram features. There are three design decisions that I encountered while working with its XMI files. These three methods cause problems in the driver and make it difficult to implement the Epsilon driver. The first one is about the link elements' target and source attributes. For some reason, the link element's target and source attributes are reversed. Target attribute stores the starting node element's *xmi:id* value and source stores the node element that link finishes (The direction indicated by the arrow). In this project, target and source access in Epsilon isn't reversed as in the XMI file. This makes usability better for users. The second problem is caused by some of the links in the GSN diagram. Most of the links such as Goal-to-Goal, Goal-to-Context, Goal-to-Solution, Strategy-to-Assumption stored as link elements with target and source attributes. However, Goal-to-Strategy and Strategy-to-Goal links didn't store as link elements. Goal-to-Strategy links are not even in the XMI file. Strategy-to-Goal links somehow in the XMI file but, they aren't in the link element form. These links are stored in the Strategy elements' *describedInference* attribute. Table 4 shows an example Strategy element. This Strategy element has three connections to three different Goal elements and these three link types are Strategy-to-Goal. It also has another link element that comes from a Goal element (Goal-to-Strategy) but this link doesn't in the XMI file. The last problem is the *xmi:id* usage. When you export the GSN diagram as XMI file, it generates a unique ID for the root element and it uses the root element's ID to generate child elements *xmi:id* attribute. However, if the user changes something in the GSN diagram (e.g. add another node), it completely changes all ID values. This is not a huge concern in the project. I cannot generate the unique *xmi:id* values for newly created elements because I don't know what values Astah GSN uses when generating these IDs.

Chapter 4 presented the design and the challenges of the Astah GSN driver development process. Also, why two of the selected Epsilon drivers chose over others and how did I take advantage of each of them while designing this project. There were also some problems and challenges while developing this project due to Astah GSN limitations. In the next chapter, the implementation progress of the chosen design layouts, as well as each developed classes and their methods, will be discussed.

5. IMPLEMENTATION

Epsilon developed with Java and all Epsilon languages run on Java. Epsilon source code consists of several Java projects. Some of the project types are EOL engine, features, plugins, and tests. The model drivers like UML and Plain-XML are plugin projects. However, each driver has more than one Java project. For instance, the Astah GSN driver consists of two different Java projects. The first one has Model features such as getters and setters, and the second one has Epsilon plugin features. The plugin project and its features didn't change much other than the plugin name variables. The crucial changes are made in the first project folder named "org.eclipse.epsilon.emc.astahgsn". Since it is a driver project, it is in the EMC (Epsilon Model Connectivity Layer) layer.

Five Java classes implemented in this project. These five classes and their content gathered from Epsilon Plain-XML driver and then their methods heavily updated by me. Names of these five classes are GsnModel, GsnProperty, GsnPropertyType, GsnPropertyGetter, and GsnPropertySetter. Classes and their methods will be explained in this section below.

5.1. GSN Model

GSN Model is the Astah GSN driver's main class. It consists of file operations, model load/store operations, all elements collector, new element creator, and element deletion methods.

XMI uses XML structure therefore, Plain-XML driver's file operations weren't changed in the Astah GSN driver. Moreover, model loading and storing functions as well as removing an element and collecting all elements functions are the same as Plain-XML driver. Most of the changes done within new element creation, and owns the function.

Plain-XML element creator function was using tag name for new elements but the Astah GSN XMI file uses the same tag name for all elements except root tag. The new function takes as a type parameter and parses it in the GsnProperty class. It returns the element's type such as Goal, Strategy, or root. Then new attributes and values are created with type data. All node and link elements have five common attributes: *xsi:type, xmi:id, id, content, and description. Content* and *description* attributes created empty. *Xsi:type* value comes from the GsnProperty class's parser function. *Id* value also comes from GsnProperty parse function but it only consists of the new element's type prefix such as G for Goal typed element. The hardest part was generating new *xmi:id* for the new element. Each element unique value and Astah GSN uses the root element's *xmi:id* to generate new *xmi:id* values for each element. Since I don't know which parameters Astah GSN uses for ID generation, I couldn't implement a working ID generator. So, instead of empty *xmi:id* values, current function puts type prefix letter + "MustBeUnique" string in the *xmi:id* attribute. For example, a new goal element's *xmi:id* attribute value will be "GMustBeUnique".

Another difference between Plain-XML and Astah GSN drivers would be appending new elements into the model. Plain-XML driver appends new elements into the model when they are created but Astah GSN doesn't append them to model directly. Appending requires another command which is ".append". This function will be explained in the setter class.

Owns function in model class responsible for calling the right class functions. For instance, if *owns* function returns falls for given input then it calls the superclass of the GsnProperty which are JavaProperty. JavaProperty class doesn't have any XML parser so, for correct elements, the *owns* function has to return true so that model class can call GsnProperty class. Two more conditions added to the *owns* function. The first one is added for getting the root element and the second one is added for list elements. These list elements are generated for getting all elements with a specified type like all goal elements. This element list should call the GsnProperty methods thus it returns true for element array lists.

5.2. GSN Property

GsnProperty is a parser class. It parses given elements and returns a newly created GsnProperty object. This class consists of a few protected attributes.

- ***GsnPropertyType*** **gsnPropertyType:** Type of the element (e.g. **Goal**)
- ***String*** **idPrefix:** Element type ID prefix (e.g. **G** for Goal).
- ***String*** **xsiType:** *xsi:type* attribute value for given type (e.g. **ARM:Claim** for Goal).
- ***boolean*** **isNode:** Is the element a node or not?
- ***boolean*** **isLink:** Is the element a link or not?
- ***boolean*** **isRoot:** Is the element the root or not?

There are also three functions in this class. Two of them are parser and the last one is element type determiner. The first parser class gets string input and parses it. This string input could be ID like G1, A4, J5, or a type name like a solution, strategy, … With these inputs, it creates a new GsnProperty object, assigns the above variables according to the element type, and returns it. If it couldn't parse the given string properly, it returns null. Elements with custom ID values return null by this parser function so they use second parser function. This parser function gets an element object as an input and parses it by *xsi:type* attribute. Nonetheless, some elements have the same values for *xsi:type*. In this case, the parser function calls the third function which is "isJustificationOrContext". This function determines if the given element Goal or Justification and also Solution or Context. As mentioned before, Context and Justification elements only connect to the Asserted Context link's target side. So, this function checks every Asserted Context elements' target attribute and if it finds the given element's *xmi:id* in them, it returns true. Otherwise, it returns false.

5.3. GSN Property Type

This class only consists of element types enumeration. Six node types plus three link types and the total nine GSN types. All element types are listed below:

- Goal
- Strategy
- Solution
- Context
- Assumption
- Justification
- AssertedInference
- AssertedEvidence
- AssertedContext

5.4. GSN Property Getter

Getter class is used for all element access queries. There are several different getter commands in the Astah GSN driver and all of them are in the invoke the function. There are major design changes in the getter class and its methods. The Plain-XML driver uses tag names to parse the XML file. For instance, "t_argumentELement.all" is an example Plain-XML query in EOL. On the other hand, Astah GSN XMI file's tag names are fixed and they don't change with element types. So, using tag names in the EOL query isn't necessary for the Astah driver. Instead, I used a "gsn" keyword to parse the file and get the root element. After the "gsn" keyword, the user can type the element he/she wants to access. For example, for accessing the goal elements the user needs to type "gsn.goal" or for accessing an assumption element with ID: A4, he/she needs to type "gsn.A4".

The major change part is the element access. The Plain-XML driver requires ".all" keyword to get all the tag elements. For example, "t_argumentElement.all" query parses the file gets all elements with *argumentElement* tag name. If you want to get the id attribute values of these tags, you need to type a query like "t_argumentElement.all.a_id". Without *all* keyword such as "t_argumentElement.a_id" the Plain-XML driver won't work. However, in the Astah GSN driver, you don't need to use the *all* keyword to access any element. For this change to happen, I need to modify the *owns* function in the GsnModel. *Owns* function returns true to all *Element* (W3C Dom library's Element class) types and also *EolModelElementType* objects with the "gsn" keyword. So, all queries that start with the "gsn" keyword invoke the GsnPropertyGetter and GsnPropertySetter classes. If the *owns* function returns false for the input object, it invokes the superclasses which are JavaPropertyGetter and JavaPropertySetter. Nonetheless, these two classes cannot parse the XMI file so invoking the right classes is a very important aspect of this driver project.

In the Plain-XML driver, ".all" request handled by the JavaPropertyGetter, not by the PlainXmlPropertyGetter. ".all" request parses the file and collects all elements with the given tag name. On the other hand, in the Astah GSN driver, ".all" requests handled by the GsnPropertyGetter class thanks to modified GsnModel's *owns* function. Without *EolModelElementType* acceptance in the GsnModel's *owns* function, Astah GSN queries have to use "all" keyword such as "gsn.all.G1" instead of "gsn.G1". Nevertheless, ".all" queries handled the same as ".goal" or ".nodes" queries thanks to this modification.

As mentioned before, the "gsn" keyword represents the root element of the XMI file. All other elements are children of the root element. Some queries like "gsn.all" or "gsn.strategy" return element Sequence which is a list type of the Epsilon languages, other queries such as "gsn.J4" or "gsn.t_g1_s_s2" return only one element object. There is also the third type of queries which returns a string as a result. Queries like "gsn.Sn2.id" or "gsn.goal.content" return String or String Sequence depending on the number of the elements.

I. All elements

Keyword: all

Returns: NULL or 1+ elements

Description: This command's input parameter is the root element. The function parses child elements and returns them as an Epsilon's sequence type.

II. All node elements

Keyword: nodes

Return Type: NULL or 1+ elements

Description: Nodes command parses the root element and creates a new list with only node elements. Node elements have non-empty ID attributes. It loops over all child elements and only adds elements with non-empty ID attribute into the result list.

III. All link elements

Keyword: links

Return Type: NULL or 1+ elements

Description: Links command works similar to nodes command. It parses root element, loop overs every child, and only adds elements with empty ID into the list. Then it returns the result list.

IV. Element by ID

Keywords: G1, Sn2, … (GSN Element ID)

Return Type: NULL or 1 element

Description: There are two types of search by ID methods: proper ID and custom ID. Getting element by ID is the last case in the invoke function. The proper ID part (e.g. G4, S2, C1) calls GsnProperty parser with a string ID variable. If it can parse it. It will return a new GsnProperty object. Then, it checks every elements' ID and if it can find it, it returns the element. Custom ID part works the same bu it only invokes this part after returning null from GsnProperty parser. If there wasn't any element with a given custom ID, invoke function returns null.

V. Element by type

Keywords: goal, strategy, solution, context, assumption, justification, assertedcontext, assertedevidence, and assertedinference

Return Type: NULL or 1+ element

Description: Element types are determined via GsnProperty parser. This part loops over every child element and calls parser to determine the element's type. If the types are a match, it adds elements to the result list. Finally, it returns the result list.

VI. Link element with source and target IDs

Keywords: s_ID1_t_ID2 or t_ID1_s_ID2

Return Type: NULL or 1 element

Description: This case takes string input like "s_G1_t_C2" and parses it to get two-node ID values. The reason I used "_" characters between each part is, Epsilon doesn't work with "–" character. After parsing sting, it finds node elements with given two IDs. If both of the nodes are found, it loops over every link element and tries to find given nodes *xmi:id*s in the target and source attributes. Finally, it returns the link element or null depending on a search result.

VII. Element's type

Keyword: gsntype

Return Type: Empty string or type string

Description: Gsn type case's input could be a root element, list, or just an element. It uses the GsnProperty element parser to determine the given element/s type and returns it. Additionally, the "gsntype" keyword used instead of the "type" keyword because the "type" keyword used by Epsilon and it returns the given object's type.

VIII. Element's target

Keyword: target

Return Type: NULL or 1+ elements

Description: Target getter works differently for node and link elements. If the element is a node, it returns the given node's all links that are targeted to the given node. If the element is a link, it returns the link's targeted node element.

IX. Element's source

Keyword: source

Return Type: NULL or 1+ elements

Description: The source case works the same as the target case. The only difference is, it checks the source attribute instead of the target attribute.

X. Element's content

Keyword: content

Return Type: Empty string or content string

Description: This case directly returns the given element/s content string.

XI. Element's ID

Keyword: id

Return Type: Empty string or ID string

Description: Returns given element/s ID value/s.

XII. Element's *xmi:id*

Keywords: xmiid or xmi_id

Return Type: Empty string or xmi:id string

Description: Returns given element/s *xmi:id* value/s.

XIII. Element's *xsi:type*

Keywords: xsitype or xsi_type

Return Type: Empty string or xsi:type string

Description: Returns given element/s *xsi:type* attribute value/s.

In addition to the element getter's invoke method, three custom methods implemented from starch by me. These three methods are

- Find element by attribute name and value (Returns the element with the given attribute name and matched value)
- Get element attribute (Returns the given attribute names value)
- Find link by node IDs (Returns the link element with target and source IDs)

5.5. GSN Property Setter

Every element value update calls use this setter class. Similar to getter class, setter class only uses one invoke function. Since all elements' values cannot change, the setter class doesn't have that many different commands.

I. Set element's content

Keyword: content

Description: Sets the given element's content attribute value.

II. Set element's ID

Keyword: id

Description: The ID command sets the given element's ID attribute.

III. Set element's *xmi:id*

Keywords: xmiid or xmi_id

Description: This command updated the given element's *xmi:id* attribute value.

IV. Set element's *xsi:type*

Keywords: xsitype or xsi_type

Description: Updated given element's *xsi:type* attribute.

V. Set the link element's target

Changes are given to the link element's target attribute. It takes an ID as a string, finds the node with the given ID, gets node's *xmi:id* attribute, and sets given link element's target attribute to the new nodes *xmi:id*.

VI. Set the link element's source

Keyword: source

Description: Works like the target setter case.

VII. Set element's GSN type

Keyword: gsntype

Description: Takes new type string as an input. It calls GsnProperty parser to find the given type's *xsi:type* value and sets it to the given element's *xsi:type* attribute.

VIII. Append a new element into the model

Keyword: append

Description: Takes an element object as an input. If the new element object doesn't have an ID with digits (e.g. G, S), it finds the highest ID number for the new element's type and assigns the highest/largest ID to the new element. Then it adds the new element into the root element as a children tag.

Similar to the getter class, the setter class has a custom method as well.

- Get the highest number of given typed element ID (Returns the given types highest ID number. For example, if the input parameter is a *goal*, it finds the highest ID goal element such as G10 and returns 10 as a result.)

The implementation chapter focused on the classes, methods and descriptions. Chapter 6 will evaluate the testing phase and testing methods. The first subject will be the user guide of the driver and the second part will investigate the driver's usability on real-world GSN diagrams.

6. TESTING & EVALUATION

In this section, usage examples are given. Each requirement would be explained with examples.

6.1. How to Use Astah GSN Driver?

6.1.1. Loading the Astah GSN model into Epsilon

Since this driver based on the Plain-XML driver, the loading model file code is the same. The only difference between these 2 drivers is names. For instance, the “Plain-XML Document” changed to “Astah GSN XMI Document”. Other than names rest of the model loading code is works like a Plain-XML driver. The below steps explains how to load an Astah GSN model into Epsilon.

a. Create a new EOL file in Eclipse IDE with Epsilon.
b. Right-click the EOL file and click *Run As* > *Run Configuration*.
c. Choose *EOL Program* and create a new Run Configuration.
d. Choose your EOL files in the *Source* tab.
e. Go to the *Model* tab and click *Add* button
f. Choose the *Astah GSN XMI Document* and click *OK* (Figure 1).

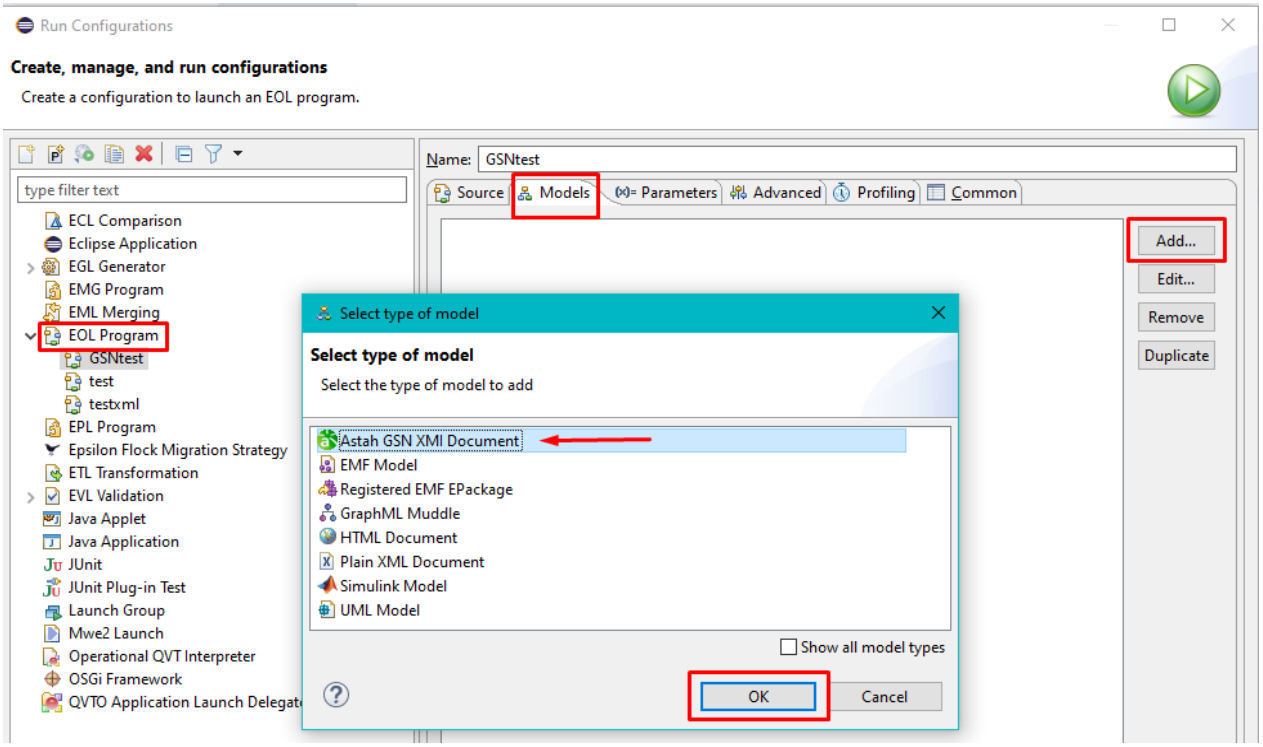


Figure 1: Loading Astah GSN model into EOL

g. Give a name to your model, it is not very important if you don't want to use multiple models in the same EOL script.
h. Choose your XMI file if it's already in the workspace. If not, add your model file into the workspace.
i. If you are going to update/modify the GSN model, choose both *Read on load* and *Store on disposal* options (Figure 2).

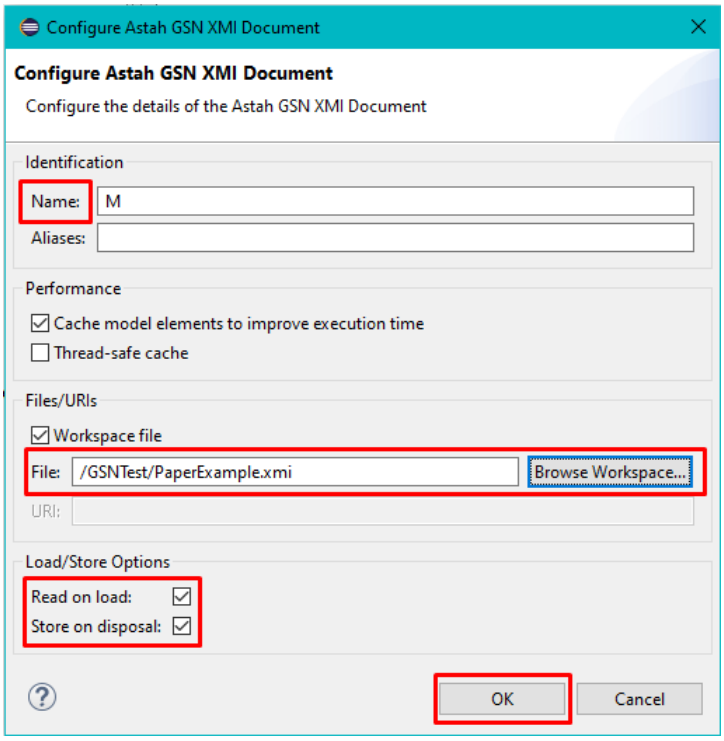


Figure 2: Loading model configurations

j. After that, you can run the EOL script with the *Run* button. EOL will run on your Astah GSN model (Figure 3).

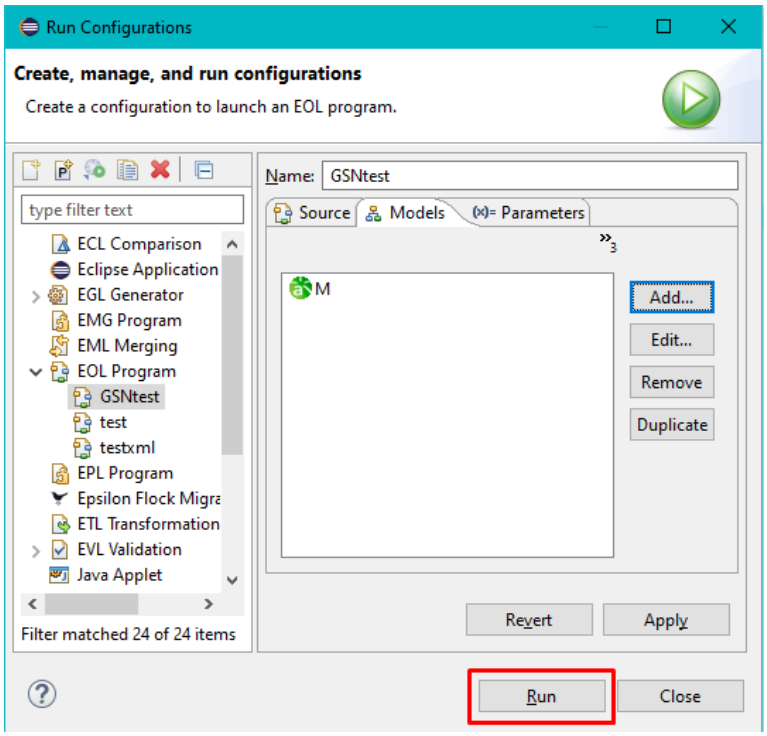


Figure 3: Running EOL with Astah GSN model

6.1.2. Reading/accessing GSN models with EOL

Plain-XML driver has 2 classes for getters and setters. Reading and accessing model calls getter class functions. Astah GSN driver's getter class completely changed and it has minimal similarities with Plain-XML getter class.

Most of the changes made for getting attribute values rather than tag names. Plain-XML getter class has a method for getting tags, child tags, attribute values, etc. In Plain-XML driver, users can only select different tag named elements but the Astah GSN XMI document requires getting different attribute valued elements.

Plain-XML documents can have several layered elements so the driver can get the root element or any child element. Astah GSN driver has two different options: getting the root element or root's child elements. Because GSN XMI document only has the root element (tag name: ARM:Argumentation) and its child elements (tag name: argumentElement). Some of the methods get the root element and some of them only get children. For instance, *gsn.all* call parses the document and returns the root element. On the other hand, *gsn.goal* or *gsn.S4* calls to parse the document, find the specified child elements, and return them as a list or a single element. Table 5 shows the attribute (value) access commands, Tables 6 and 7 present single and multiple element access commands respectively.

Attribute Getter Commands	Command Explanation

`gsn.all.content`
`gsn.context.content`
`gsn.Sn13.content`
	Returns specified element/s' content attribute value/s. The results could be Sequence or string depending of an element. Link elements' don't have content attribute so it returns empty string ("").

`gsn.G1.gsntype`	Returns given element/s' GSN type
`gsn.strategy.id` `gsn.c23.id`	Returns given element/s' ID. Link elements' don't have ID thus it returns empty string.
`gsn.a9.xmi_id`	Returns given element/s' xmi:id attribute. Each element has unique xmi:id values.
`gsn.goal.xsi_type`	Returns given element/s' xsi:type attribute. Same elements (e.g. goal and assumption) have the same xsi:type values.
`gsn.Sn3.target` `gsn.t_g1_s_g2.target`	Returns link elements that have target value as given node element/s.
`gsn.g2.source` `gsn.s_g2_t_g1.source`	Returns link elements that have source value as given node element/s.

Table 5: Astah GSN Driver Attribute Getters

Single Element Commands	Command Explanation
`gsn.G1, gsn.c12, gsn.S3, gsn.Sn7, gsn.a5, gsn.j9`	Returns node element with given ID value.
`gsn.CA1, gsn.ERC1 (For ID: E-RC1)`	Returns node element with given custom ID value. *Note: Custom ID queries cannot include non-alphanumerical characters.*
`gsn.t_g1_s_g2 OR gsn.s_g2_t_g1`	Returns link element that has target value is G1 and source value is G2.
`gsn.all.last`	Returns the last element of the given element list
`gsn.solution.first`	Returns the first element of the given element list

Table 6: Astah GSN Driver Single Element Getters

Multiple Element Commands	Command Explanation
`gsn.all`	Returns all elements (Entire model)
`gsn.nodes`	Returns all node elements
`gsn.links`	Returns all link elements
`gsn.goal`	Returns all Goal elements
`gsn.strategy`	Returns all Strategy elements
`gsn.solution`	Returns all Solution elements
`gsn.context`	Returns all Context elements
`gsn.assumption`	Returns all Assumption elements
`gsn.justification`	Returns all Justification elements
`gsn.assertedcontext`	Returns all Asserted Context (link) elements
`gsn.assertedinference`	Returns all Asserted Inference (link) elements
`gsn.assertedevidence`	Returns all Asserted Evidence (link) elements

Table 7: Astah GSN Driver Multiple Element (List) Getters

Some of the getters could be combined differently. For example, *gsn.goal.last.content.println()* and *gsn.goal.content.last.println()* prints the same result. The difference between these 2 commands is simple. The first command gets all goal elements list, then finds the last goal element and prints its content attribute value. The second command gets the goal elements list, then gets all goal elements contents and creates a new list later it prints the last content in that list. Thus, the first command is faster than the second one because it doesn't get all goal elements' content attribute, it just gets one goal element's content attribute and prints it.

6.1.3. Updating GSN models with EOL

Updating elements command call setter class functions. Most of the element attributes can be set via the below commands. Getter and setter commands are the same. The only difference is setters require "=" character and new value after equals character. Table 8 indicates a few element update commands in Astah GSN driver.

Setter Commands	Type	Command Explanation
`gsn.sn13.content = "New content";`	String	Updates the given element's content value
`gsn.g3.id = "G6";`	String	Updates the given element's ID (ID attributes must be unique!)
`gsn.j15.xmi_id = "fvLpH5q4Eeqyz11T9RpXrQ";`	String	Updates given element's xmi:id attribute. However, Astah GSN generates unique xmi:id values based on model and element location. Therefore, using this command can corrupt the model file.
`gsn.c7.xsi_type = "ARM:ArgumentReasoning";`	String	Updates given element's xsi:type attribute. Changing xsi:type without changing id might corrupt model file.
`gsn.t_s1_s_g1.target = gsn.sn7;`	Node element	Updates the given link element's target attribute to the new node element's xmi:id. The new value must be a node element.
`gsn.t_J1_s_G13.source = gsn.J2;`	Node element	Updates the given link element's source attribute to the new node element's xmi:id. The new value must be a node element.
`gsn.a12.gsntype = "goal";`	String	Updates the given element's type attribute. Changing xsi:type without changing id might corrupt model file.

Table 8: Astah GSN Driver Element Setters

6.1.4. Creating new elements in GSN model with EOL

Creating a new element command uses a *new* keyword. Using a *new* keyword-only creates a new element object but it doesn't append this new object into the model. Table 9 demonstrates a new element creation and attachment to model commands in EOL.

Creator Command	Command Explanation
`var newElement = new goal;`	The *new* keyword creates a new element with the given type.
`gsn.all.append = newElement;`	*Append* command attaches a given element into the model file. A new element would be the last element in the model file.

Table 9: Astah GSN Driver Element Creator Commands

The new element's attributes could be set in two different ways. Either updating the *newElement* object like *`newElement.content = "test";`* or accessing the last element and updating its attributes such as *`gsn.all.last.content = "test";`*.

6.1.5. Deleting elements in GSN model with EOL

Deleting an element in EOL uses a *delete* keyword. One or multiple elements could be deleted via *delete* command. Table 10 shows the element removal command in the driver.

Delete Command	Command Explanation
`delete gsn.G10;`	Deletes given element/s.

Table 10: Astah GSN Driver Element Delete Commands

6.1.6. Epsilon Validation Language (EVL) usage

Running EVL or other Epsilon language script in Astah GSN models requires the same steps as running EOL scripts. First, you need to load the model, choose the Astah GSN XMI file, and create a new run configuration for EVL to run it. However, if you have an EGX file, you should choose the EGX file's path as a source to run.

In this example, two different constraints added to the model. The first constraint covers all elements in the model so, its context is "gsn". This one checks the G1 element's outgoing links. If

the G1 element doesn't have exactly 2 outgoing links, it will give an error and will print the error message.

The second constraint is only for goal elements so, context is "goal'. This one checks every goal elements' content attribute and if one of them has an empty content value, it prints the given error message. These two examples are constraints thus they will print messages in red. However, EVL also supports warning messages as well and the user can choose between these two options. Listing 1 displays the example EVL script on GSN models.

```
context gsn {
   critique g1MustHaveTwoOutgoingLinks {
      check: self.g1.source.size() == 2
      message: "Goal " + self.g1.id + " must have exactly 2 outgoing
targets!"
      }
}

context goal {
   constraint emptyGoalContent  {
      check{
         // Create a new sequence for storing Goal IDs
         var emptyGoals : Sequence;
         // Loop over every Goal elements
         for(g in self.goal){
            // If the Goal element has empty content value
            if(g.content == ""){
               // Add its ID in the sequence
               emptyGoals.add(g.id);
            }
         }
         // If there is at least one Goal element with empty
         // content, give an error (return false)
         if(emptyGoals.isEmpty){
            return true;
         }
         else{
            return false;
         }
      }
      // Show each Goal elements' ID that has empty content
      message: "Goals " + emptyGoals + " must have non-empty
content!"
   }
}
```

Listing 1: EVL usage example in Astah GSN driver

Table 11: EVL usage example in Astah GSN driver

6.1.7. Epsilon Code Generation Language (EGL) usage

This EGL and EGX scripts will generate an HTML table for all Goal elements in the Astah GSN model. Table's rows will contain Goal IDs and Goal contents. EGX script handles file operations such as read and creates a newly generated file. EGL script handles code generation operations. In this EGL, I used for loop to go over each goal element and putting their ID and content values inside the HTML table's rows. Listing 2 shows the EGX and Listing 3 indicates the EGL scripts, respectively.

```
pre { "Transformation starting".println(); }
rule AstahGSN2HTML
      transform gsn : GSN {
            template : "YourEGLFileName.egl"
            target : "output.html"
      }
post { "Transformation finished".println(); }
```

Listing 2: EGX usage example in Astah GSN driver

```
<table border="1">
      <tr><td>Goal ID</td><td>Content</td></tr>
      [%for (g in gsn.goal.sortBy(g|g.id)){%]
      <tr>
            <td>[%=g.id%]</td>
            <td>[%=g.content%]</td>
      </tr>
      [%}%]
</table>
```

Listing 3: EGL usage example in Astah GSN driver

6.1.8. Epsilon Transformation Language (ETL) usage

Epsilon Transformation Language example requires two different models and metamodels. Thus instead of shoving the ETL script in this section, the transformation of the GSN-to-EMF example will be discussed in the next section.

6.2. GSN Model Examples with Epsilon

GSN diagrams mostly used in safety-critical systems development. Thus, I tried to use real-world safety-critical GSN diagrams within the Epsilon as a project example. Thanks to Dr. Paige, and Workflow+ project students Nicholas Annable and Thomas Chiang, I gathered three different GSN models. The first GSN model is from GSN Standards Document and it is a basic GSN model for a safety-critical system. The second one is from Thomas Chiang's coffee cup safety model. And the last one is from Workflow+ project.

Different Epsilon languages and queries will be used on these three models.

6.2.1. GSN Community Standard Model

GSN Community Standard Model is an example model from GSN Community Standard Version 1 document [6]. This document explains the GSN basics, how and where to use it and the model structure. Figure 4 displays the GSN Community Standard Model diagram.

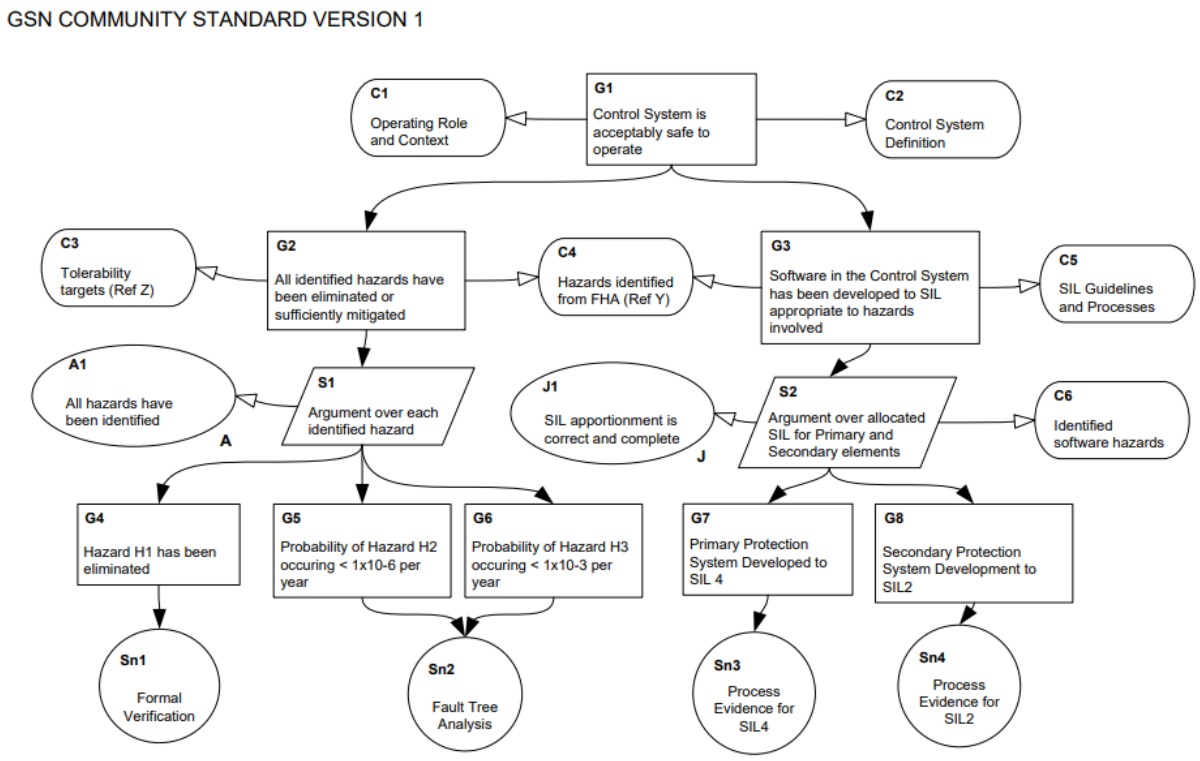


Figure 4: An Example GSN Diagram from GSN Standards Document

6.2.1.1. EOL Examples on GSN Community Standard Model

Epsilon Object Language helps users to access or update the model elements. Update operations include changing element values, creating new elements and deleting existing elements from the model. Table 14 demonstrates several element queries in the Astah GSN driver.

Commands	Results
`gsn.all`	All elements (nodes+links)
`gsn.solution`	4 Solutions elements: Sn1-Sn4
`gsn.justification`	J1 Justification element
`gsn.goal.size`	8 – Number of Goals
`gsn.C6.content`	"Identified software hazards"
`gsn.t_G3_s_G1`	G1-G3 link (Asserted Inference) element
`gsn.s_g8_t_sn4.target.id`	"G8" - G8-Sn4 link (Asserted Evidence) element's targeted node ID
`gsn.strategy.last`	S2 strategy element
`gsn.c4.xsi_type`	"ARM:InformationElement" – Context element's xsi:type value
`gsn.Sn1.content = "Informal Verification";`	Sets Sn1 Solution elements content to "Informal Verification"
`var newStgy = New Strategy;`	Creates a new Strategy element
`gsn.all.append = newStgy;`	Generates a new ID for a new element (S3) and appends it to the model
`delete C6;`	Deletes ID=C6 Context element

Table 12: Element Queries on GSN Community Standard Model

6.2.1.2. EVL Examples on GSN Community Standard Model

EVL is used for model validation operations. In this part, 2 different model validation examples shown on the GSN Community Standard model with Astah GSN driver operations. One of the examples is warning and the other one is an error type constraint. If the given Astah GSN XMI file (model) doesn't validate for these 2 constraints, it will show the error message in the Epsilon console.

The first constraint validation for Standard Community Model is:

- Every Strategy element must have at least one Context connection.

This constraint is just for showing Astah GSN driver's capabilities, normally GSN models don't require Strategy-to-Context connections. This validation method was chosen as a warning so it is a critique instead of a constraint. Table 15 shows the constraint validation in EVL.

```
context strategy {
   critique StrategyWithoutContext{
      check{
         // Boolean variable for does strategy element have a Context
         // connection or not
         var doesHaveContext = false;
         // List (Sequence) for holding all Strategy elements' ID for
         // warning message.
         var strategyIdWithoutContext : Sequence;
         // For loop over all Strategy elements
         for (stgy in self.strategy) {
            // Reset boolean value for each strategy element
            doesHaveContext = false;
            // For loop over each Strategy elements' each outgoing
            // target links
            for (StrategySources in stgy.source) {
               /* If Strategy has outgoing links AND
               * If outgoing link element's target side node element's
               * type is Context, change the boolean value to TRUE
               */
               if(not (StrategySources == null) and
                  StrategySources.target.gsntype == "Context")
               {
                  doesHaveContext = true;
               }
            }
            // If Strategy element doesn't have any connection to
            // Context, add its ID into the list
            if(doesHaveContext == false){
               strategyIdWithoutContext.add(stgy.id);
            }
         }
         /* After checking every Strategy element, it found some
         * Strategies without Context connection --> Give a warning and
         * print message.
         */
         if(strategyIdWithoutContext.isEmpty()){
            return true;
         }
         else{
```

```
            return false;
         }
      }
      message: "Strategy " + strategyIdWithoutContext + " should have
an connection to a Context element."
   }
}
```

Listing 4: StrategyWithoutContext Validation on GSN Community Standard Model

The second validation example tries to detect Solution elements that don't have any (incoming) connections.

- Solutions must have at least one incoming link (connection).

It checks every Solution element and if they don't have an incoming connection link, it adds their IDs in the list to print the error message. Table 16 indicates the EVL code for this constraint.

```
context solution {
   constraint SolutionMustHaveConnection {
      check{
         // Sequence for storing Solution IDs for error message
         var solutionsWithoutConnection : Sequence;
         // Loop over every solution element
         for(sol in self.solution){
            // Check if the Solution has an incoming (target)
            // connection or not
            if(sol.target == null){
              // If target is empty for the given Solution, add its
              // ID in list
               solutionsWithoutConnection.add(sol.id);
            }
         }
         // Check if there are Solutions without connection or not
         // If the list is empty, don't give an error
         if(solutionsWithoutConnection.isEmpty()){
            return true;
         }
         else{
            return false;
         }
      }
      message: "Solutions " + solutionsWithoutConnection + " must
have a connection!"
   }
```

```
}
```

Listing 5: SolutionMustHaveConnection Validation on GSN Community Standard Model

6.2.1.3. EGL Examples on GSN Community Standard Model

Running EGL scripts in Epsilon requires EGX files for filenames. Below EGX script runs given EGL file (GSNModelToHTMLTable.egl) and generates the HTML output file (output.html). It also prints "Transformation starting/finished" before and after the EGL operations. Table 17 shows the EGX script and Table 18 shows the EGL script to generate HTML tables form the GSN model.

```
pre { "Transformation starting".println(); }
rule AstahGSN2HTML
      transform gsn : GSN {
      // EGL file name
      template : "GSNModelToHTMLTable.egl"
      // Output HTML file name
      target : "output.html"
post { "Transformation finished".println(); }
```

Listing 6: HTML Table Generator EGX Script for GSN Community Standard Model

The below code shows the EGL scripts for generating HTML tables from Astah GSN model XMI files. It generates an HTML table with Goal ID, Goal's number of outgoing links and Goal' s number of incoming links. The output of this script could be seen below the EGL code.

```
<table border="1">
      <tr>
            <td>Goal ID</td>
            <td># Outgoing Connections</td>
            <td># Incoming Connections</td>
      </tr>
      [%for (g in gsn.goal.sortBy(g|g.id)){%]
      <tr>
            <td>[%=g.id%]</td>
            <td>[%=g.source.size()%]</td>
            <td>[%=g.target.size()%]</td>
      </tr>
      [%}%]
</table>
```

Listing 7: HTML Table Generator EGL Script for GSN Community Standard Model

The output of the EGL script is shown in Table 19. It is not the same number of connections as the GSN model due to XMI file link storage types. It doesn't store Strategy-to-Goal and Goal-to-Strategy connections as a link element.

- G2 has 5 outgoing link elements: G2-C3, G2-C4, G2-G4, G2-G5 and G2-G6.
- G3 has 4 outgoing link elements: G3-C4, G3-C5, G3-G7 and G3-G8.

Goal ID	# Outgoing Connections	# Incoming Connections
G1	4	1
G2	5	1
G3	4	1
G4	1	1
G5	1	1
G6	1	1
G7	1	1
G8	1	1

Table 13: GSN Model to HTML Table EGL Result

6.2.1.4. ETL Examples on GSN Community Standard Model

In this section, the GSN model will be transformed into a Structured Content model. Figure 5 shows the two models and transformed parts. All GSN model's nodes transformed to table rows. Nodes' ID, xsi:type and content attributes transformed to Structured Content's cells. Transformation ETL script is shown in Table 20 and its output Structured Content model's XML file is shown in Table 21.

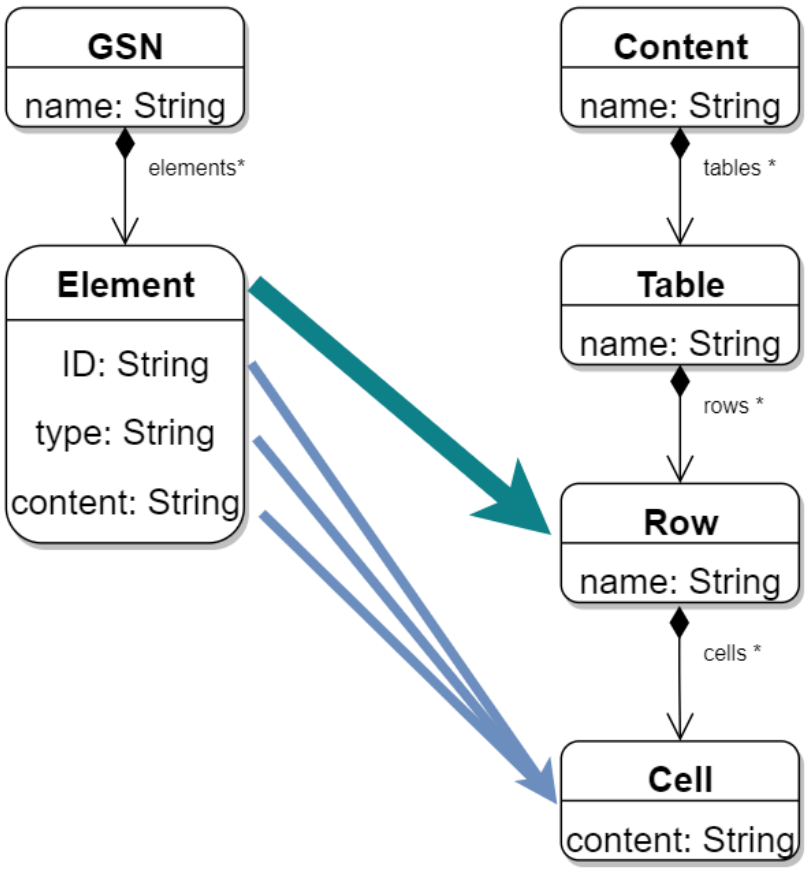


Figure 5: Astah GSN XMI Model to Structured Content Model Transformation

```
pre {
	"Transformation starting".println();
}

rule GSN2EMF
	// Source is Astah GSN XMI Model file
	transform a : Source!GSN
	// Target is EMF Structured Element empty model
	to t : Target!Table {
	// Name of the table -> GSN Table
	t.name = "GSN Nodes Table";
	// Table header row
	t.createRow(Sequence{"Element ID", "Element Type", "Content"});
	// Loop over all nodes and create a new row for each one
	for (g in Source!gsn.nodes.sortBy(g|g.id)) {
		t.createRow(Sequence{g.id, g.xsi_type, g.content});
	}
}
```

```
// Create rows in the table
operation Target!Table createRow(content : Sequence) {
      var row : new Target!Row;
      for (c in content) { row.createCell(c); }
      self.rows.add(row);
}

// Create cells in the rows
operation Target!Row createCell(content : Any) {
      var cell : new Target!Cell;
      cell.content = content + "";
      self.cells.add(cell);
}

post {
      var root : new Target!Content;
      root.tables.addAll(Target!Table.all);
      "Transformation finished".println();
}
```

Listing 8: GSN to Structured Content ETL Script for GSN Community Standard Model

The output Structured Model XML file shows the first four rows of the table. Each row represents different node elements and each cell stores these nodes ID, type and content values.

```
<?xml version="1.0" encoding="ASCII"?>
<structuredContent:Content xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:structuredContent="http://www.example.org/structuredContent"
  xsi:schemaLocation="http://www.example.org/structuredContent
  StructuredContent.ecore">
  <tables name="GSN Nodes Table">
    <rows>
      <cells content="Element ID"/>
      <cells content="Element Type"/>
      <cells content="Content"/>
    </rows>
    <rows>
      <cells content="A1"/>
      <cells content="ARM:Claim"/>
      <cells content="All hazards have been identified"/>
    </rows>
```

```
    <rows>
      <cells content="C1"/>
      <cells content="ARM:InformationElement"/>
      <cells content="Operating Role and Context"/>
    </rows>
    <rows>
      <cells content="C2"/>
      <cells content="ARM:InformationElement"/>
      <cells content="Control System Definition"/>
    </rows>
    <rows>
      <cells content="C3"/>
      <cells content="ARM:InformationElement"/>
      <cells content="Tolerability targets (Ref Z)"/>
    </rows>
       …
```

Listing 9: ETL Script's Output Structured Content Model XML File

6.2.2. Coffee Cup Safety Standards GSN Model

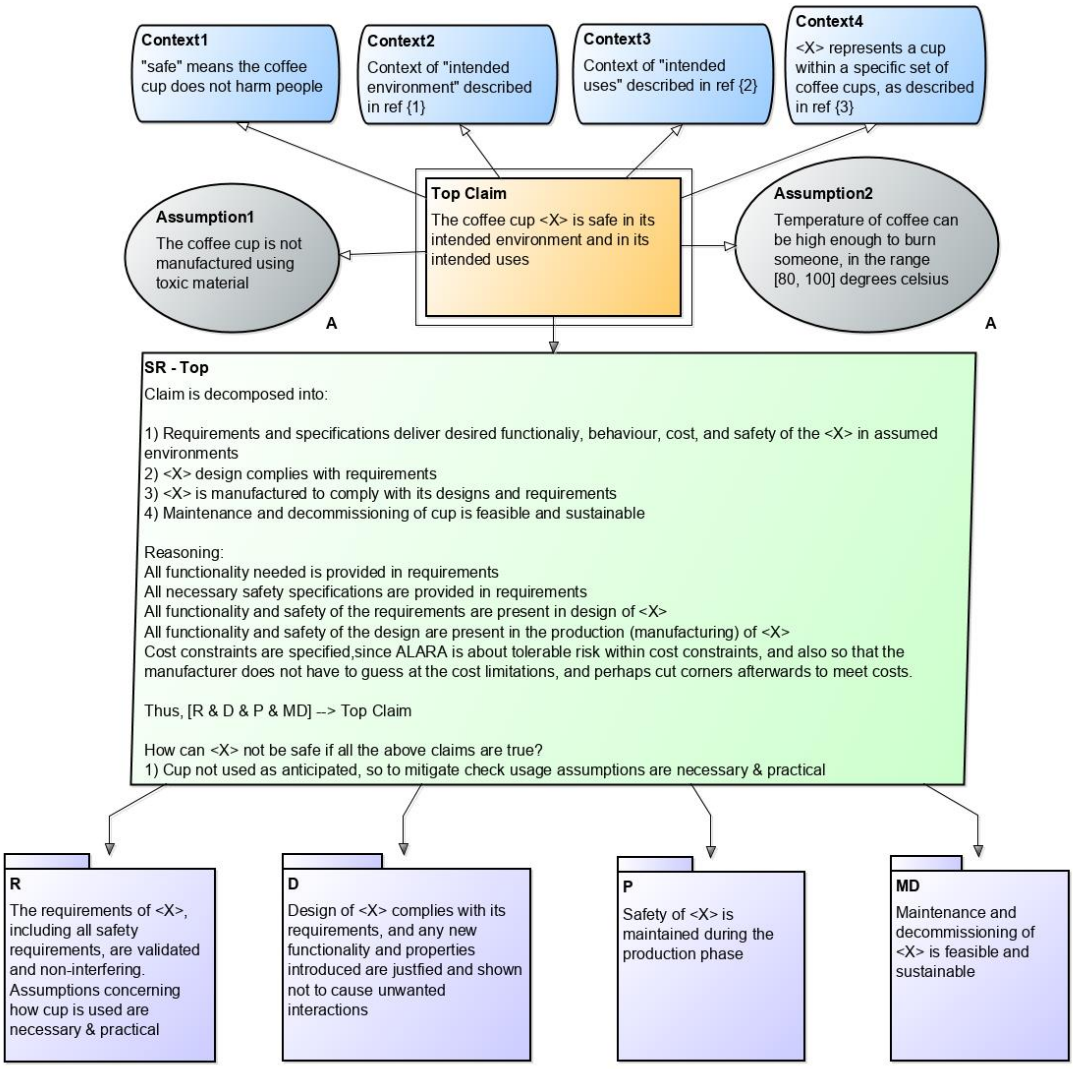
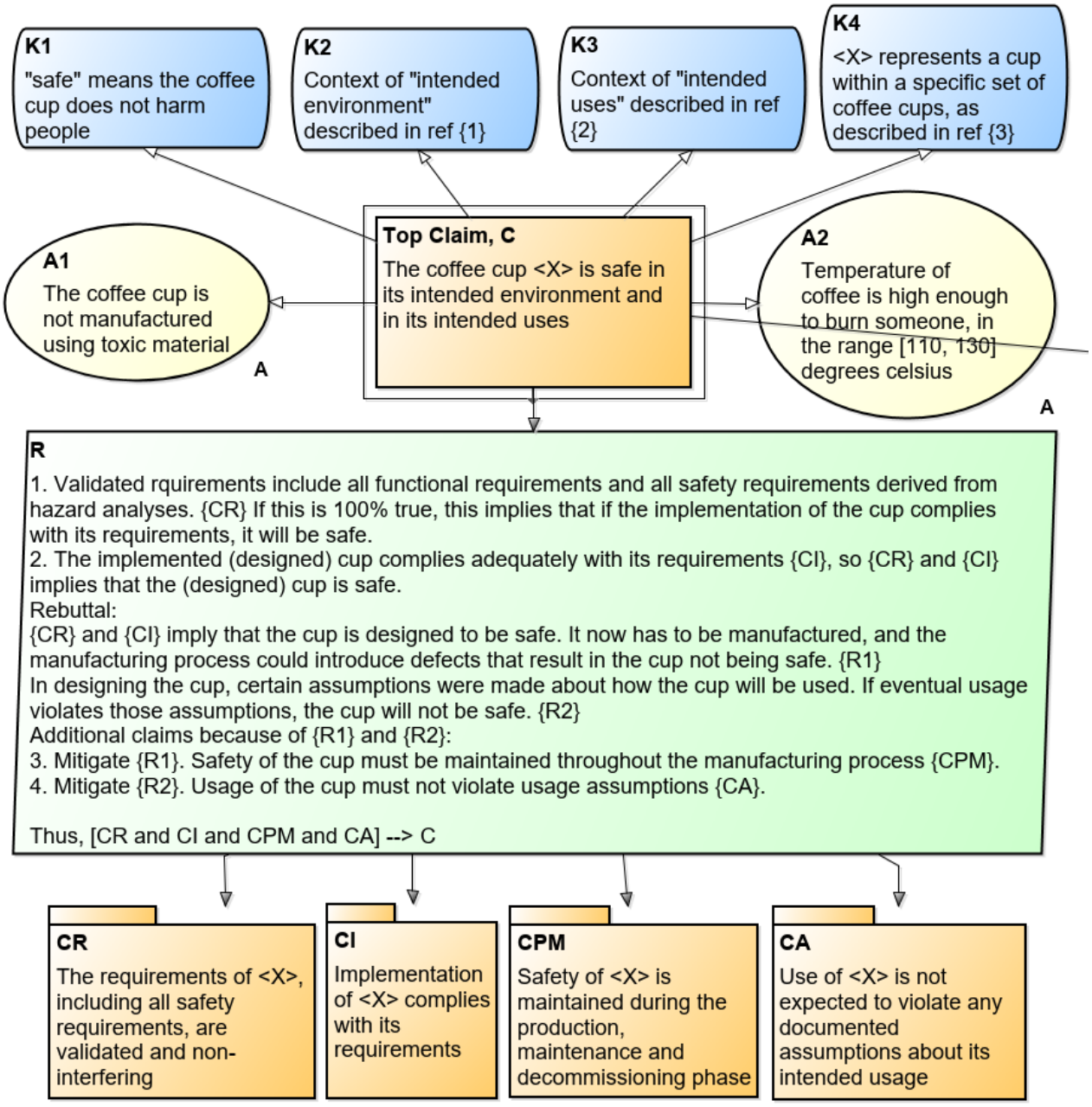


Figure 6: An Example GSN Diagram for Coffee Cup Safety Cases


K1
"safe" means the coffee cup does not harm people
K2
Context of "intended environment" described in ref {1}
K3
Context of "intended uses" described in ref {2}
K4
<X> represents a cup within a specific set of coffee cups, as described in ref {3}
Top Claim, C
The coffee cup <X> is safe in its intended environment and in its intended uses
A1
The coffee cup is not manufactured using toxic material
A
A2
Temperature of coffee is high enough to burn someone, in the range [110, 130] degrees celsius
A
R
1. Validated rquirements include all functional requirements and all safety requirements derived from hazard analyses. {CR} If this is 100% true, this implies that if the implementation of the cup complies with its requirements, it will be safe.
2. The implemented (designed) cup complies adequately with its requirements {CI}, so {CR} and {CI} implies that the (designed) cup is safe.
Rebuttal:
{CR} and {CI} imply that the cup is designed to be safe. It now has to be manufactured, and the manufacturing process could introduce defects that result in the cup not being safe. {R1}
In designing the cup, certain assumptions were made about how the cup will be used. If eventual usage violates those assumptions, the cup will not be safe. {R2}
Additional claims because of {R1} and {R2}:
3. Mitigate {R1}. Safety of the cup must be maintained throughout the manufacturing process {CPM}.
4. Mitigate {R2}. Usage of the cup must not violate usage assumptions {CA}.
Thus, [CR and CI and CPM and CA] --> C
CR
The requirements of <X>, including all safety requirements, are validated and non-interfering
CI
Implementation of <X> complies with its requirements
CPM
Safety of <X> is maintained during the production, maintenance and decommissioning phase
CA
Use of <X> is not expected to violate any documented assumptions about its intended usage

6.2.3. Workflow+ Metamodel

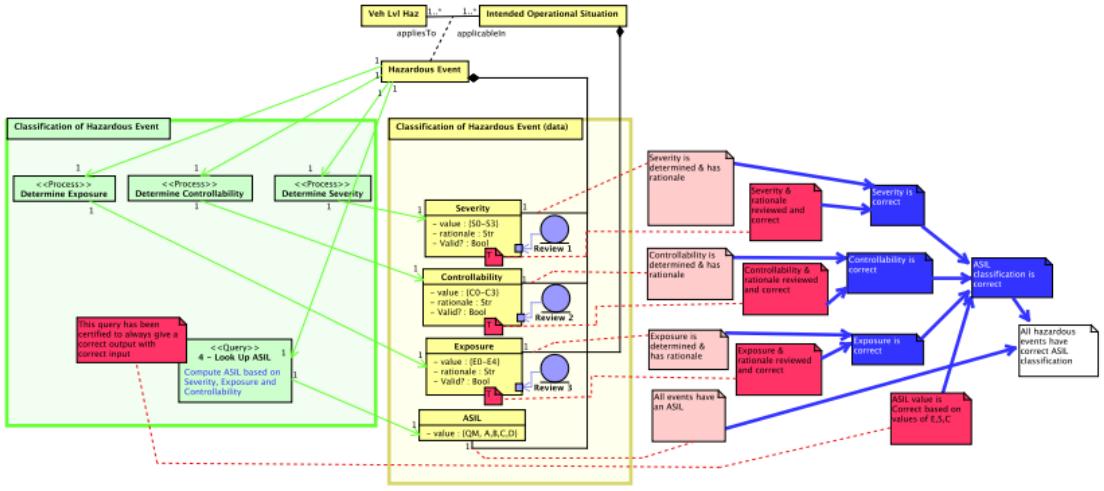


Figure 4.10: Workflow+ metamodel of classification of hazardous events with argumentation as described in ISO 26262-3 [29]

Figure 7: Workflow+ metamodel of classification of hazardous events with argumentation as described in ISO 26262-3

7. CONCLUSION

Conclusion…

7.1. Future Work

This project is a Master of Engineering project and the project time is not limited. Thus, it doesn't have all the features that I can implement into the Astah GSN driver. For instance, the current version of the driver cannot access or update the Strategy-to-Goal and Goal-to-Strategy link elements due to their storage type in the XMI file. This feature could be implemented in the driver but changing the design of the Astah GSN XMI file would be better for all. Also, for newly created elements unique *xmi:id* values needed. However, without knowledge of what Astah GSN uses in the ID generation phase, generating new unique IDs might corrupt the whole GSN model. Thus, implementing this feature would require additional information and it might take longer. Moreover, accessing and changing all attributes in the XMI file could be implemented in the driver if necessary.

A. REFERENCES

[1]. Schmidt, Douglas C., 'Model-Driven Engineering', Computer-IEEE Computer Society, vol. 39, no. 2, pp. 25-31, 2006.

[2]. Epsilon, 'Eclipse Epsilon', 2020. [Online]. Available: https://www.eclipse.org/epsilon. [Accessed: 04- Jul- 2020].

[3]. EpsilonLabs, 'Epsilon-HTML Integration', 2020. [Online]. Available: https://github.com/epsilonlabs/emc-html. [Accessed: 04- Jul- 2020].

[4]. GSNAdmin, 'What is the Goal Structuring Notation?', 2020. [Online]. Available: https://www.goalstructuringnotation.info/archives/category/in-a-nutshell. [Accessed: 04- Jul- 2020].

[5]. Object Management Group (OMG), 'XML Metadata Interchange (XMI) Specification', 2020. [Online]. Available: https://www.omg.org/spec/XMI/2.5.1/PDF. [Accessed: 04- Jul- 2020].

[6]. The GSN Working Group Online, 'GSN Community Standard Verison 1', 2011. [Online]. Available: http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf. [Accessed: 08- Jul- 2020].