CHALLENGES IN IMPLEMENTING AN EPSILON DRIVER FOR ASTAH GSN

By

BARAN KAYA, M.Eng.

A Project Report
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:	v, 48

ABSTRACT

Model-Driven Engineering (MDE) methods are increasingly used for Safety-Critical software systems development. Goal Structuring Notation (GSN) diagrams are one of the most used models for arguing for acceptable safety of systems; GSN is becoming a standard for safety argumentation purposes. Epsilon is an Eclipse-based platform for model management, providing several programming languages for different forms of model manipulation, including transformation, generation of text, and validation. Epsilon is also modelling technology-agnostic: it provides facilities that allow Epsilon programs to work on any kind of model with the right driver. Astah GSN is one of the most advanced pieces of commercial software for working with GSN diagrams, however, there is no Epsilon driver for Astah GSN. In this project, a new Epsilon driver for Astah GSN was developed. While developing this driver, technical and design challenges were encountered. This report explains the challenges that were encountered, and demonstrates the use of the driver in a number of examples, showing how different Epsilon languages can be applied to GSN models.

CONTENTS

1. INTRODUCTION

Goal Structuring Notation (GSN) is a de facto standard used for specifying safety and assurance arguments. The construction of a GSN model is an important step when preparing for a safety-critical system to be certified. However, with safety-critical systems growing in size and complexity, it is becoming increasingly difficult to manage GSN models. Automated support for managing – transforming, validating, generating documentation – GSN models would be beneficial to engineers.

Epsilon is an open-source model management platform, supporting a variety of different model management tasks, including validation and different forms of transformation. Epsilon is also applicable to many different modelling technologies (e.g., EMF, XML, spreadsheets) via its *connectivity layer*. More specifically, in principle Epsilon can manage any type of model but requires individual drivers for each model type.

Astah GSN is one of the most popular commercial full-featured GSN tools, but it lacks support for model management. The focus of this project, which is informally in collaboration with General Motors, is to provide Epsilon driver support for Astah GSN models. In turn, this will allow GSN models to be programmatically manipulated using Epsilon programs.

Even though Epsilon is an open-source platform, implementing a driver for a commercial tool like Astah GSN is more complicated than is typically the case for open-source tools. For example, Astah GSN's XMI export feature lacks consistency (as we discuss in later chapters), and it was impossible to look at the source code for Astah GSN to understand why this was the case, and how to resolve it. Figuring out these problems and solving them were key challenges that had to be overcome in this project. Although most of the challenges of building an Astah GSN driver for Epsilon have been solved, some of them remain. All challenges encountered while developing the project, design choices, implementation of the Java classes and the testing methods (with examples) are explained in this project report.

The report is organized as follows: Section 2 describes related work on Model-Driven Engineering, safety engineering and the Epsilon framework. Section 3 specifies the project's requirements, including technical requirements for the driver. In Section 4, the design of the driver and the encountered challenges presented. Section 5 explains the implementation, including a brief overview of each Java class of the project. In Section 6, testing methods and evaluation techniques are given. Also, two example GSN models are used to demonstrate some of the capabilities of the driver. Lastly, Section 7 summarizes the project and suggests future work.

2. RELATED WORK

This project, in collaboration with General Motors, aims to provide Epsilon driver support for Astah GSN models. By doing so, Epsilon programs (for transformation, validation, querying, impact analysis, etc) can be applied to both Astah GSN models and heterogeneous sets of models (including Astah GSN models). GM uses Model-Driven Engineering (MDE) methods and GSN models in the process of developing safety-critical systems. In this project, MDE, safety-critical system development and Epsilon terminologies were used thus, an overview of related work in these areas is given in Sections 2.1 to 2.3. Firstly, MDE and why it is used will be explained in section 2.1. Then, in section 2.2 safety engineering, assurance cases and GSN diagrams will be clarified. Finally, in section 2.3 the Epsilon platform, EOL, EMC and challenges in developing a new driver for Epsilon will be explained.

2.1. Model-Driven Engineering

Model-Driven Engineering (MDE) is becoming a standard for software development in a number of software industries. MDE aims to help the development team to reduce the complexity of the software development process via abstraction. With large software systems, even the entire development team cannot fully understand the whole system architecture and design. But abstraction can help the whole team (including non-developers) to understand the basic software design. Models apply abstraction on software development based on two key principles: decomposition and mapping [1]. Decomposition involves breaking large problems (and large designs) into parts, while mapping involves the transformation of one abstraction into a related abstraction. These two key principles are inherent in MDE approaches to software development.

Besides the abstraction features, MDE also aims to make it easier to develop a software system with a combination of domain-specific modelling languages (DSML) and transformation engines/generators [2]. Safety-critical systems are large and very complex and they consist of several sub-systems. Developing large systems with general-purpose languages (GPL) is getting harder, especially if the system has to be safety-critical. Unlike GPLs, DSMLs help developers to design better and safer systems within their domains. Avionics, transportation and many others use DSML to create safer software systems. Transformation engines and generators aim to help programmers transform models into different models and generate source code or documents from the designed models. Code generation from the model has the potential to eliminate boilerplate code and the ability to transform the model to another model provides flexibility in development. These MDE methods arguably can reduce software development time and cost.

2.2. Safety Cases & Goal Structuring Notation

Safety is the condition of being protected from causing danger, injury or risk. System safety is the application of technical engineering and management processes to optimize safety in a system. A safe system treats accidents as a control problem rather than a failure; also, it tries to impose restrictions on system actions and operations [3]. The concepts of safety and reliability are often mixed across software systems. As Leveson and Moses [4] state, reliable systems could be unsafe; on the other hand, safe systems could be unreliable. These two properties are not dependent on each other. A system is a safe system if it can handle component failures and accidents [3]. Designing these kinds of systems is challenging, and it often involves the creation and evaluation of a *safety case*. Kelly [5] defines safety cases as follows: "A safety case should communicate a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context".

Ensuring the safety of the software system is a hard and complex job. Safety engineering uses system constraints to arrange safety in the system itself. Enforcing and finding out these constraints while designing the system is a very crucial step in safety engineering. There are different methods to do so and one of the commonly used ones is STAMP (System-Theoretic Accident Model and Processes) [3]. STAMP tries to reduce component failures and component interaction failures by enforcing the safety constraints to the system [3]. However, STAMP was developed for general safety-critical systems not specifically for software systems. Thus, it lacks some aspects of the development of software-intensive systems. On the other hand, STPA (System-Theoretic Process Analysis) [6] is developed for safety-critical software development. It fills the gaps between safety-critical software development methods and STAMP. STPA is an early hazard analysis technique. This technique tries to detect early design flaws in both software and hardware systems [6]. STPA contains two steps: determining potential system hazards and identifying their occurrence possibilities [3]. With STPA, designing a safe system from scratch is intended to be easier for designers and safety engineers.

A safety-critical system should be safe from the beginning. Thus, safety provision starts in the early design phases; it is harder to make an already designed system safer than designing a safer system from scratch. Safety cases are a very crucial part of the safety-critical system design process. Each safety-critical industry has its standards for safety besides every safety case and its evidence must be objective and meet the safety requirements [7]. Many safety cases are expressed in a textual format, as a lengthy document with many sections and subsections covering risk, process stages, types of evidence, and cross-references between sections. Using a text-based or document-based format for defining safety cases might be appropriate and understandable for smaller systems but the bigger the system the more complex its safety cases become. Goal Structuring Notation (GSN) is a graphical approach to representing and modelling safety cases. It is widely used in the safety-critical industries, particularly in avionics and aerospace. GSN represents every argument and their relationships [8]. It is easier to see the safety claims and their evidence in the GSN diagrams rather than text-based ones. Designing the system is just the beginning of the system's lifecycle. Safety-

critical systems require maintenance like any other system. As shown in the [9], using GSN diagrams can potentially make it easier to maintain the system and safety case change process.

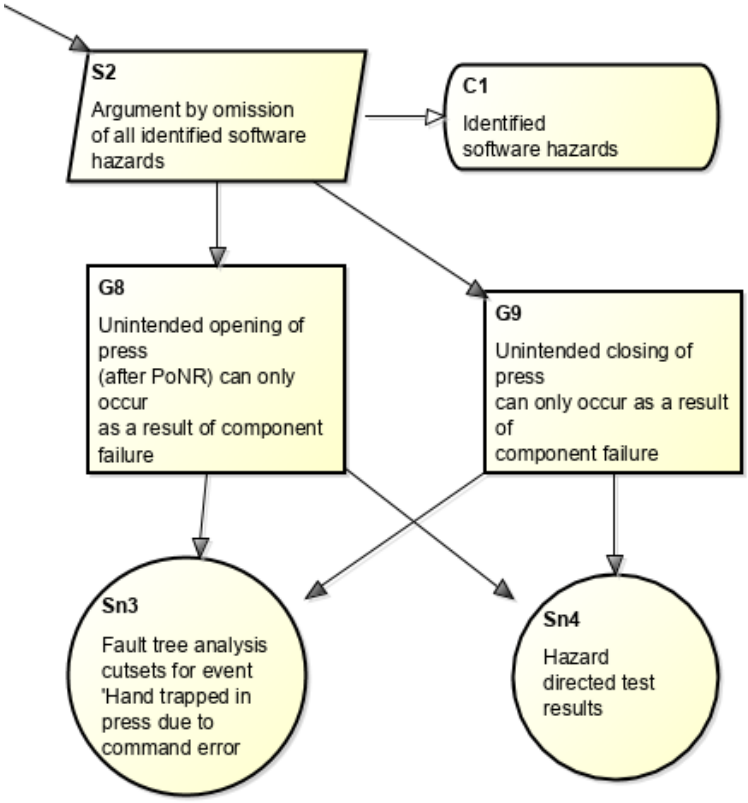

Figure 1: GSN Diagram Example

Figure 1 shows part of an example GSN diagram with two goals (G8, G9), two solutions (Sn3, Sn4), one strategy (S2) and a context (C1) element. GSN shows the claims (goals) and the evidences (solutions) and their relationships. Between claims and sub-claims, there can be strategy elements for the reasoning step. For indicating claims or reasonings context, context elements could be used. Additional assumptions and justifications can be added to GSN.

Astah GSN is one of the most full-featured commercial GSN diagram tools. It has several features for easier usability. These features vary from diagram styling to different types of file export. For instance, the alignment feature makes it easier to edit the diagram's shape end elements' position. Different export or import filetypes such as XMI, diagram image or MindMap increase Astah GSN's usability with other model management tools. Last and one of the important features is model validation and fix. Astah GSN can validate drawn models and can fix them with a click.

2.3. Epsilon

Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) is a platform for model management operations such as validation, transformation or code generation [10], [11]. Epsilon has ten different languages for different model management tasks. Epsilon Object Language (EOL) is the base language for all other task-specific Epsilon languages [12]. All other languages inherited from EOL and they all use EOL syntax for reducing repetition and better intelligibility [11]. Kolovos et al. [11] state that EOL's main aim is to be a core language for *higher-level task-specific* languages, however; it can also be used for model access and modification operations. That's why developing an Epsilon driver for EOL basics is adequate.

Epsilon is modelling technology agnostic. This means that all languages in Epsilon, including EOL, can be used to access and manipulate models in a variety of modelling technologies, including EMF, XMI, spreadsheets, Simulink and numerous others. This is enabled by the Epsilon Model Connectivity (EMC) layer, which is based on the driver design pattern. Every modelling technology that is desired to be manipulated by Epsilon requires a driver. EMC abstracts from all different models using the IModel interface [11]. The Epsilon Book [11] explains that the EMC layer handles the file and model operations such as model loading, type&ownership operations and creation/deletion/modification operations on the model. All these operations could and should be customized for each modelling technology. Some of the model connectivity drivers such as Plain-XML or UML model come with Epsilon installation. However, for other models, users need to create a new driver or may check the Epsilon Labs GitHub repository for the driver.

Epsilon supports most of the common models like UML, EMF, XML and many others. Nonetheless, it doesn't support every existing model. Drivers are needed to work on models in the Epsilon platform and many commercial modelling tools don't have a driver for the Epsilon. Most of the commercial tools are not open-source thus developing Epsilon drivers for these tools is harder. This causes a technological gap between closed-source model tools and open-source model management tools [13]. Open-source model management tools like Epsilon are developing day by day but commercial tools are mostly stuck in the same position by their developers' support. Using open-source model management tools instead of commercial model tools built-in management tools would benefit the company that is using these. However, developing an Epsilon driver for a commercial tool is a challenging task and it might take a serious amount of time. For instance, as demonstrated in [13], developing a driver for Rolls-Royce's safety-critical systems is very challenging and the driver requires high-performance management for very big models.

This project is at the intersection of Model-Driven Engineering and safety, focusing on bridging commercial and open-source tools. This will be carried out by developing an Epsilon driver for Astah GSN. Astah GSN is a commercial tool and does not provide an easy-to-use API for programmatically and externally manipulating models. Hence, it is useful to build an Epsilon driver for Astah GSN. As we will see, building this atop existing drivers, especially the XML driver for Epsilon, will allow us to produce a quality solution within the time constraints of this project.

3. REQUIREMENTS

The main goal of the project is to be able to use Epsilon to manage Astah GSN models. To support this, an Epsilon driver for Astah GSN must be produced. This driver will need to be able to parse Astah GSN correctly. Astah GSN saves its model files with a '.agml' extension and this is a proprietary format that cannot be parsed and manipulated directly. However, it also provides XMI import/export features for GSN models, and it is through this that we will develop an Epsilon driver.

Developing an Epsilon driver involves implementing a number of interfaces from the Epsilon Model Connectivity (EMC) layer. By doing so, this allows Epsilon programs written in the core language (Epsilon Object Language, EOL) to manipulate Astah GSN models. As a side-effect, because of the architecture of Epsilon, this also means that the ten task-specific languages of Epsilon can manipulate Astah GSN. This is because the task-specific languages are derived from EOL thus, integrating models to EOL is sufficient for all other language support.

The requirements of this driver project have been collected by careful analysis of the capabilities of Astah GSN and the technical design of Epsilon (particularly EMC). Listed requirements were generated by the author after examining GSN diagrams, Astah GSN XMI files, similar Epsilon drivers, and Epsilon source code. Of particular importance were element access requirements, which are based on GSN standards; and attribute access requirements, which are based on both GSN and Astah GSN XMI files. 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 element.

The requirements for this project are, therefore, entirely technical and focus on the system to be implemented, rather than on user requirements.

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

2. Users must be able to 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 different types of elements
      2.2.1. Accessing to all nodes
      2.2.2. Accessing to all links
      2.2.3. Accessing different types of nodes (e.g. goals, strategies, solutions, …)
      2.2.4. Accessing different types of links (e.g. asserted evidence)
   2.3. Users should be able to access a specific element in a GSN model
      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 node element that is the ending point of the specific link
    2.4.7. Access a node element that is the starting point of the specific link

3. Users should be able to update 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 in a GSN model with EOL.

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

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

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

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

In this chapter, project requirements and how/where they are collected were explained. The next chapter will analyze the design decisions of the project and what kind of changes/updates were made to the Astah GSN driver for Epsilon while developing it.

4. DESIGN

Designing an Epsilon driver from scratch is very challenging. It takes a lot of time but fortunately, Epsilon has lots of built-in and external drivers for several different modelling technologies. It was easier to start with a similar model driver's source code, on which the Astah GSN driver could be constructed. The most similar drivers that could be used in the design of the Astah GSN driver were the internal Plain-XML driver and external HTML driver. While designing the Astah GSN driver, these two drivers and their features helped me a lot. Both drivers and how they are used in this project will be explained in this chapter.

While developing the driver and the project, the waterfall methodology was used informally (i.e., in terms of gathering requirements, design, implementation), but some changes were made while developing the project due to uncertainties. The first step of the project was examining the GSN diagrams and learning the basics of their syntax. The next step was looking for similar Epsilon drivers, their element access operations and how they go about parsing the model files. Then project requirements were created based on collected information from GSN and similar drivers. The next step was looking for Epsilon driver projects to build upon. 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 on examples. In this step, Epsilon ran in debug mode and each step of model processing was investigated. Some necessary updates made in the source code and all of the changelog can be found in the project's GitHub repository.

The Plain-XML driver comes with the default 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 could make 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 ran 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 allows users to load, read and update XML models in Epsilon. However, after some experimentation, I determined that this driver is not useful for Astah GSN XMI files. Astah GSN XMI files store every element in the GSN model with the same tag name (*argumentElement*), but the 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 would need to be heavily modified for parsing 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 [14] GitHub page and then has to run Eclipse from the source to use it. For the Astah GSN driver project, using the HTML driver as a base project made developing a driver plugin for Epsilon easier for me. 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 source code as the Plain-XML driver. HTML driver parses HTML files and calls the Plain-XML driver's methods. That means HTML driver doesn't have any custom methods for model operations, it just uses Plain-XML driver's functions. At the end of the day, HTML is a subset of XML and using the Plain-XML driver methods would make sense. However, that means using the HTML driver source code as a base project is not useful due to a lack of custom model management functions. So instead, I used the HTML driver for its Epsilon plugin features. Every name in the HTML driver's plugin packages changed to "Astah GSN" thus, new project packages for the Astah GSN project created. More details about HTML will be explained later in Section 4.3.

4.1. Implementing an Epsilon Driver for a Commercial Tool

There are several Goal Structuring Notation diagram tools out there. However, this project was developed for General Motors and they were using Astah GSN for their main GSN diagram tool. Moreover, Astah GSN has many 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 proprietary. That's why I couldn't use the Astah GSN model files (.agml) directly 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 as a result, it's hard to identify element types. Also, the exported XMI file doesn't store the GSN diagram (the concrete syntax), it only stores elements (abstract syntax). 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. Nonetheless, not storing the diagram is a huge downside for using XMI files instead of AGML files. Another negative aspect of using XMI is that it does not store every link element. For example, it doesn't store Goal-to-Strategy and Strategy-to-Goal link elements in the file. Instead, Astah GSN stores Goal-to-Strategy-to-Goal relationships as Goal-to-Goal links and records these link elements' *xmi:id* in the Strategy elements' *describedInference* attribute. These kinds of defects make it challenging to develop an Epsilon driver for Astah GSN. We discuss these challenges further in the next sections.

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 [15] such as GSN. Astah GSN could export 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, and even content are stored in the element attributes. An XML parser is by definition able to parse XMI files, but it will be unaware of its semantic content, particularly the types of elements. More specifically, Epsilon's 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 the capability to parse by attribute features. The new Astah GSN driver developed in this project provides attribute parser and other additional features for Astah GSN models. With this driver, the user should be able to access or update each elements' attributes by 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	• Asserted Inference (goal-to-goal) • Asserted Evidence (goal-to-solution) • Asserted Context (goal-to-context, goal-to-assumption, goal-to-justification, strategy-to-context, strategy-to-assumption and strategy-to-justification)

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 (tag name: argumentElement) attributes 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. 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. EpsilonLab's GitHub page [14] has several EMC drivers for different models such as HTML, JDBC, JSON, and more. The most similar model driver to Astah GSN was HTML because HTML and XML/XMI share a tree-like structure.

After cloning the EMC-HTML git repository, I tried to run it on the Epsilon source code. The first step was importing the two HTML project packages into Eclipse Epsilon workspace. Even though the HTML repository has six different packages, only two of them are necessary for running the HTML driver. The other four 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 the HTML driver. The new driver creates an Epsilon model selection in the Run Configuration of every Epsilon language. 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 had no experience with Epsilon plugin development, I used the 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 examined 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 methods only call Plain-XML getter and setter functions. Therefore, I used Plain-XML drivers functions for the getter, setter and model classes. This is an important point to clarify: the HTML and Plain-XML drivers are interdependent. In Astah GSN, both are therefore used to support necessary functionality, though the high-level structure of the driver is derived from the HTML driver.

I used an HTML driver for plugin features and changed all models, getter, and setter classes/methods. The HTML driver used Java Jsoup library [16] for HTML element parsing. This library could also parse the XML/XMI files but each time this library is used, it adds <html> and other main tags into the XML file. So, I changed this library to use the Plain-XML driver's parser library.

4.4. Using the Epsilon Plain-XML Driver

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

The 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 [17] 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 in Chapter 5.

The model class is the main class of the driver. It is responsible for file operations, model operations, invoking getter and setter classes, element creation, and removal operations. This class wasn't modified 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 four types for XML files. These are *tag, attribute, reference,* and *child*. But these types are not necessary for the Astah GSN driver. Thus, types are changed to GSN model elements such as goal, strategy, solution, etc, as these are more readable (in the source code) and the resulting code will hopefully be easier to maintain.

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 most important parts of the Astah GSN Driver. In an early design, each user query was 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 are created by Astah GSN with element type data; specifically, goal element IDs start with G, Solution with Sn and Context with C. However, element IDs don't have to start with element type letters. In this case, the GSN type parse function for custom IDs such as CA1, AR-C1 returned null and Epsilon showed an error.

In later designs, a 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 cannot 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. Of course, checking every element in the model requires more time but there is no other way to find out if the custom ID being queried is in the model or not.

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' type 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 two element types is their connections. Goal elements can connect all three 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 types of element examples could be seen in Table 4.

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"/>
Context element
<argumentElement xsi:type="ARM:InformationElement" xmi:id="_fvLpE5q4EeqyzooT9RpXrQ" id="C1" description="" content="Operating Role and Context" url=""/>
Solution 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's XMI import/export feature is relatively immature compared to its GSN diagram features. There are three design issues that I encountered while working with its XMI files. These three methods led to complications in the driver and made it difficult to implement the Epsilon driver. The first issue 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 issue 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 aren't stored as link elements. Instead of this method, Astah GSN stores Goal-to-Strategy-to-Goal links as a link element (Goal-to-Goal) and stores this link element's *xmi:id* in-between strategy element's *describedInference* attribute. Table 4 shows an example Strategy element. This Strategy element (S1) has four connections to four different Goal elements but XMI file has three different link elements. Rather than G1-to-S1, S1-to-G2, S1-to-G3 and, S1-to-G4 links, XMI file stores G1-to-G2, G1-to-G3 and, G1-to-G4 links. It also has these three link elements' *xmi:id* inside S1's *describedInference*.

The last issue 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 but 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. It also explained why two of the selected Epsilon drivers are chosen over others and how I took 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

This chapter presents an overview of the implementation of the Astah GSN driver, focusing on integration with the existing Epsilon system (via the EMC driver facility).

Epsilon was developed with Java, and all Epsilon languages run on Java. Epsilon's source code consists of several Java projects. Some of these project types are EOL engine, features, plugins, and tests. It also includes several model drivers, like UML and Plain-XML, which are Eclipse plugin projects. However, each driver consists of 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. This is a standard architecture for Eclipse projects. The plugin project and its features can be adopted wholesale from Plain-XML directly; the only change required is to the name of the plugin variables.

The crucial new implementation features are in the first project folder named "org.eclipse.epsilon.emc.astahgsn", part of the EMC (Epsilon Model Connectivity Layer), and it is on these features and the corresponding implementation classes and methods that we concentrate in this chapter. Five Java classes are implemented in this project. These five classes and their content have been inspired by the Epsilon Plain-XML driver; the behaviour of the methods has been heavily updated by me. The names of these five classes are GsnModel, GsnProperty, GsnPropertyType, GsnPropertyGetter, and GsnPropertySetter. These classes and their important methods will be explained in the next subsections.

5.1. GSN Model Class

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

Recall once again that the Astah GSN driver builds on the Plain-XML driver of Epsilon, and Astah GSN uses XMI as a persistence mechanism. XMI uses the structure of XML; 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 in the Plain-XML driver. Most of the changes that were required were done within new element creation, and the *owns* function.

The 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 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 their values are created based on the 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 has a 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 is appending new elements into the model. The 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 more detail when we explain the setter class.

The *owns* function in the model class is responsible for calling the right class functions. For instance, if *owns* function returns false for given input then it calls the superclass of the GsnProperty which is 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. One more condition added to the *owns* function. It is added for getting the root element from the XMI file so that if an Epsilon program returns the root element, the *owns* function returns true.

5.2. GSN Property Class

The GsnProperty class 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 methods and the last one is element type determiner. The first parser class gets a string input and parses it. This string input could be an object ID like G1, A4, J5, or a type name like a solution, strategy, … With these inputs, the parser 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 Class

This class only consists of element type enumerations, constituting 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 Class

The 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 function. When compared with the Plain-XML drier, 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 users can type the element they want to access. For example, for accessing all goal elements the users need to type "gsn.goal" or for accessing an assumption element with ID: A4, they need to type "gsn.A4".

The major change is in 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 -superclass- 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 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.

V. 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.

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 the string, 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: 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: xsi_type

Return Type: Empty string or xsi:type string

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

In addition to the getter invoke method, three custom methods have been implemented from scratch 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 Class

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

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

IV. Set element's *xsi:type*

Keywords: xsi_type

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

V. Set the link element's target

Keyword: target

Description: This command changes the given 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 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.)

This concludes the overview of the implementation challenges for the Astah GSN driver, particularly focusing on how it was built using the Plain-XML driver of Epsilon. 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. EVALUATION

This chapter serves two purposes. First, it explains how an end-user might use the Astah GSN driver (and in doing so I demonstrate how a number of the requirements have been satisfied). Secondly, I show how to use Epsilon to manage several examples of Astah GSN models, in terms of querying and updating GSN models, checking constraints on models, and applying transformations (model-to-text and model-to-model) on GSN models. The GSN examples are meant to be representative of patterns seen in real or realistic GSN models taken from industrial examples.

6.1. How to Use Astah GSN Driver?

In this section, I give a short overview of how to use the Astah GSN driver to support model management of GSN models via Epsilon.

6.1.1. Loading the Astah GSN Model into Epsilon

Since the Astah GSN driver is based on the Plain-XML driver, the loading model file operation is the same. The only difference between these two drivers is their names. For instance, the "Plain-XML Document" changed to "Astah GSN XMI Document". Other than its name, the rest of the model loading operation works like a Plain-XML driver. The below steps explains how to load an Astah GSN model into Epsilon, and to execute an EOL program (e.g., a query) against the model.

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 2).

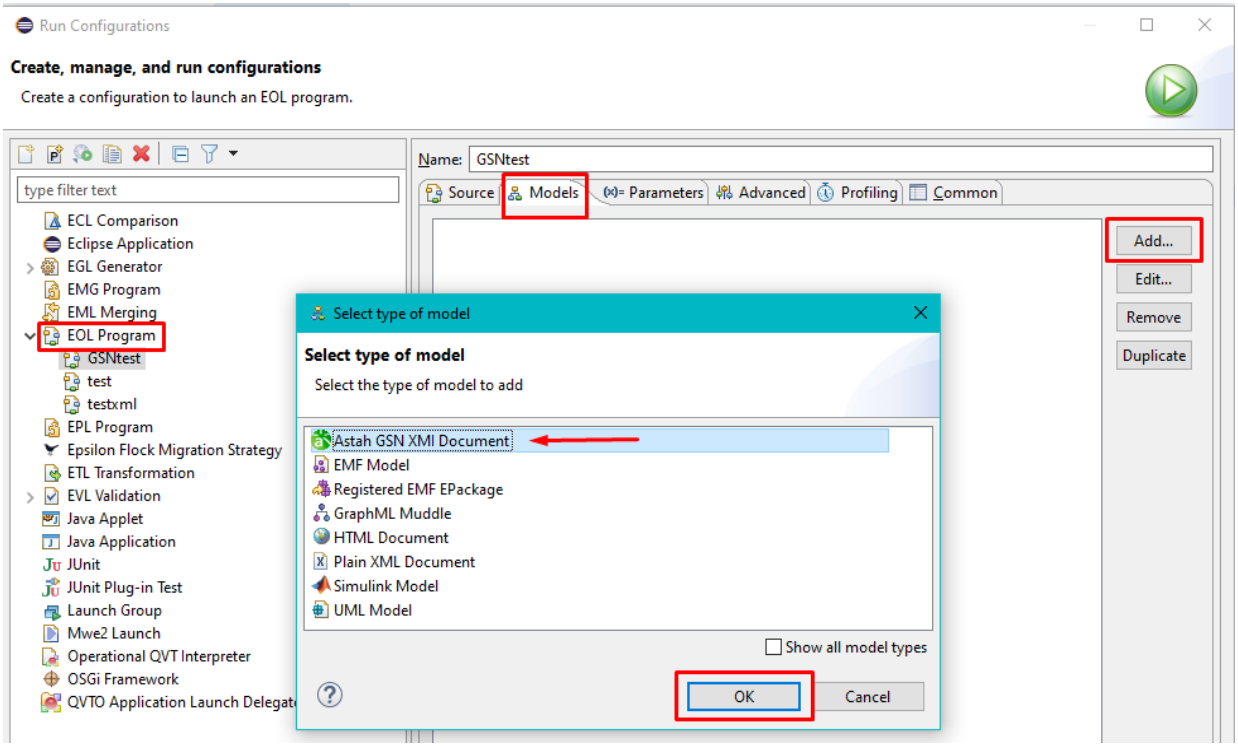

Figure 2: Loading Astah GSN model into EOL

g. Give a name to your model; this 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 3).

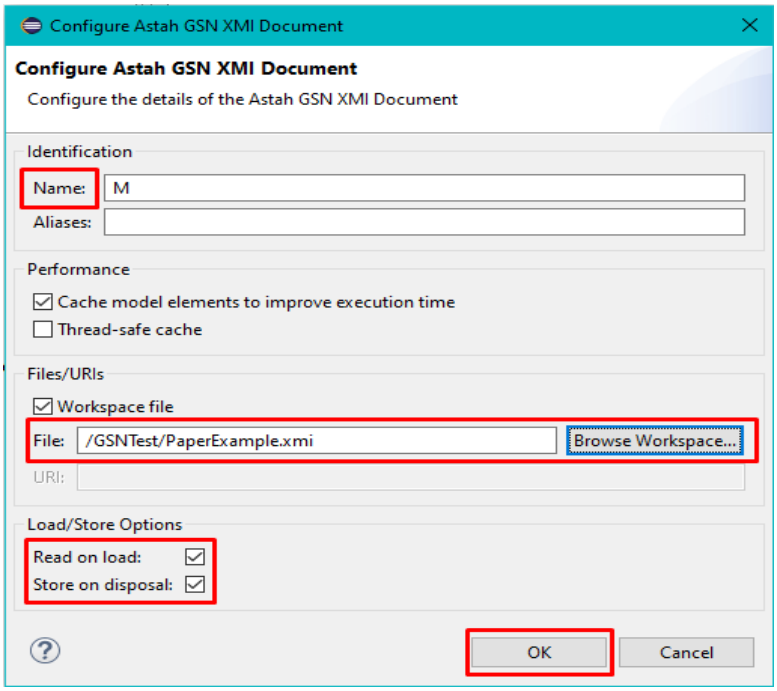

Figure 3: Loading model configurations

j. After that, you can run the EOL script with the *Run* button. EOL will run on your Astah GSN model (Figure 4).

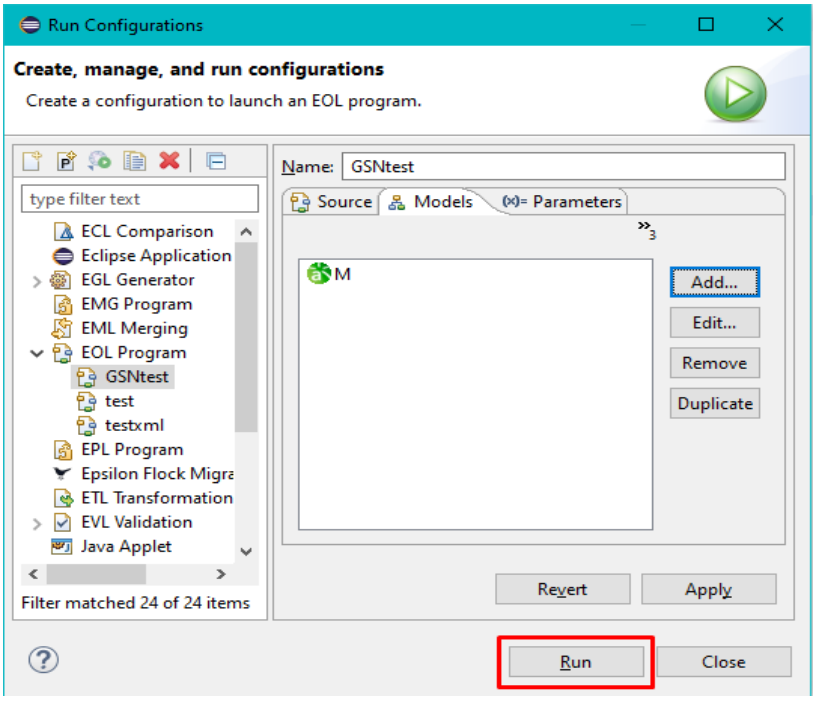

Figure 4: Running EOL with Astah GSN model

6.1.2. Reading/Accessing GSN Models with EOL

The Plain-XML driver has two 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* call parses the document and returns the root element. On the other hand, *gsn.goal* or *gsn.S4* queries parse the document, find the specified child element/s, 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 two 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 will involve calling 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 selection of the element update commands available in the 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 a 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

In previous sections, we have shown how to execute core Epsilon EOL programs against Astah GSN models. The Astah GSN driver allows *any* Epsilon program in any language to be executed against an Astah GSN model. We demonstrate this firstly by showing how to execute constraints in EVL against Astah GSN.

Running EVL (or other Epsilon language programs) against 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.

In the following example, two different constraints are checked against a model. The first constraint covers all elements in the model so, its context is "gsn". This example checks the G1 element's outgoing links. If the G1 element doesn't have exactly two outgoing links, it will give an error and will print an error message.

The second constraint is only for goal elements so its 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

6.1.7. Epsilon Code Generation Language (EGL) Usage

The driver allows model-to-text transformations to be executed against Astah GSN models. The following EGL and EGX scripts will generate an HTML table for all Goal elements in the Astah GSN model. The table's rows will contain Goal IDs and Goal contents. The EGX script handles file operations such as read and creates a newly generated file. The 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

We now show how to execute a model-to-model transformation against an Astah GSN model. This Epsilon Transformation Language (ETL) example requires two different models and metamodels. Thus instead of showing 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 are mostly used in safety-critical systems development. Thus, I tried to use real-world GSN diagrams as project examples. Thanks to Dr. Paige, and the Workflow+ project students Nicholas Annable and Thomas Chiang, I gathered two different GSN models. The first GSN model is from the GSN Standards Document and it is a basic GSN model for a safety-critical system. The

second example is from Thomas Chiang's coffee cup safety model, which is a non-confidential example used for illustration purposes in a number of theses and papers.

I will show the use of different Epsilon languages and queries on these two models.

6.2.1. GSN Community Standard Model

GSN Community Standard Model is an example model from GSN Community Standard Version 1 document [18]. This document explains the GSN basics, how and where to use it and the model structure. Figure 5 displays the GSN Community Standard Model diagram.

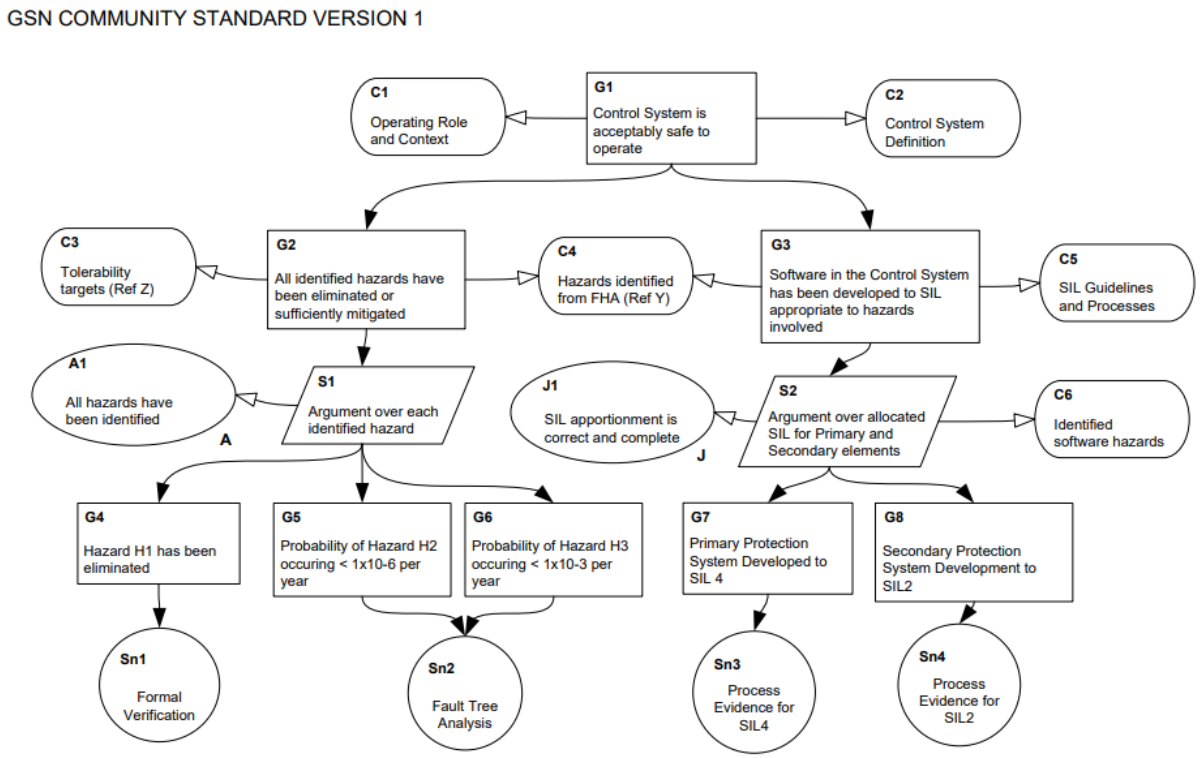

Figure 5: An Example GSN Diagram from GSN Standards Document

6.2.1.1. EOL Examples on GSN Community Standard Model

We can apply the Epsilon Object Language (EOL) to the community standard model, which can help 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 11: 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. Here, two different model validation examples are shown applied to the GSN Community Standard model, making use of Astah GSN driver operations. One of the examples is a warning and the other one is an error type constraint. If the given Astah GSN XMI file (model) doesn't validate for these two 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).

The EVL program 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 shows 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 to specify and acquire filenames. The EGX script below runs a specified EGL file (GSNModelToHTMLTable.egl) and generates an HTML output file (output.html). It also prints "Transformation starting/finished" before and after the EGL operations are executed. Listing 6 shows the EGX script and Listing 7 shows the EGL script to generate HTML tables form the GSN model. (Note that this is a standard pattern with model-to-text transformation in Epsilon: EGL files contain the behaviour of the transformation rules, while EGX files are used for coordination of the transformation workflow.)

```
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 code below 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 a Goal's number of incoming links. The output of this script can 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 12. 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 12: 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 6 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 Listing 8 and its output Structured Content model's XML file is shown in Listing 9.

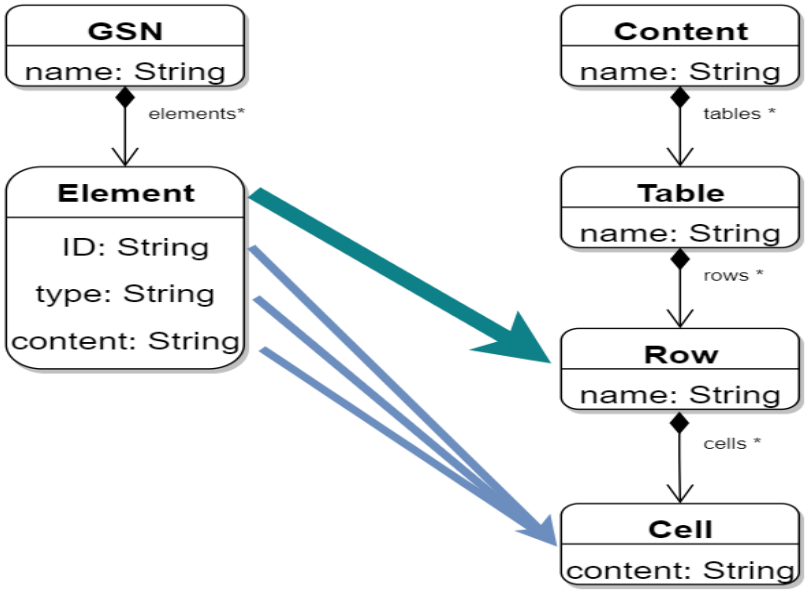

Figure 6: Astah GSN XMI Model to Structured Content Model Transformation

```
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 {
	// Append new tables in root content
	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

The Coffee cup GSN diagram appears in another previously published McMaster master's thesis [19]. This project contains different safety cases for a single coffee cup. Goal structuring notation used to illustrate safety cases. GSN is one of the most popular options for safety case management. Figure 7 shows the top-level GSN diagram for coffee cup safety cases. It has four modules: R

(Requirements), D (Design), P (Manufacturing) and MD (Maintenance). Figure 8 displays the R module and Figure 9 shows the D module top-level diagrams.

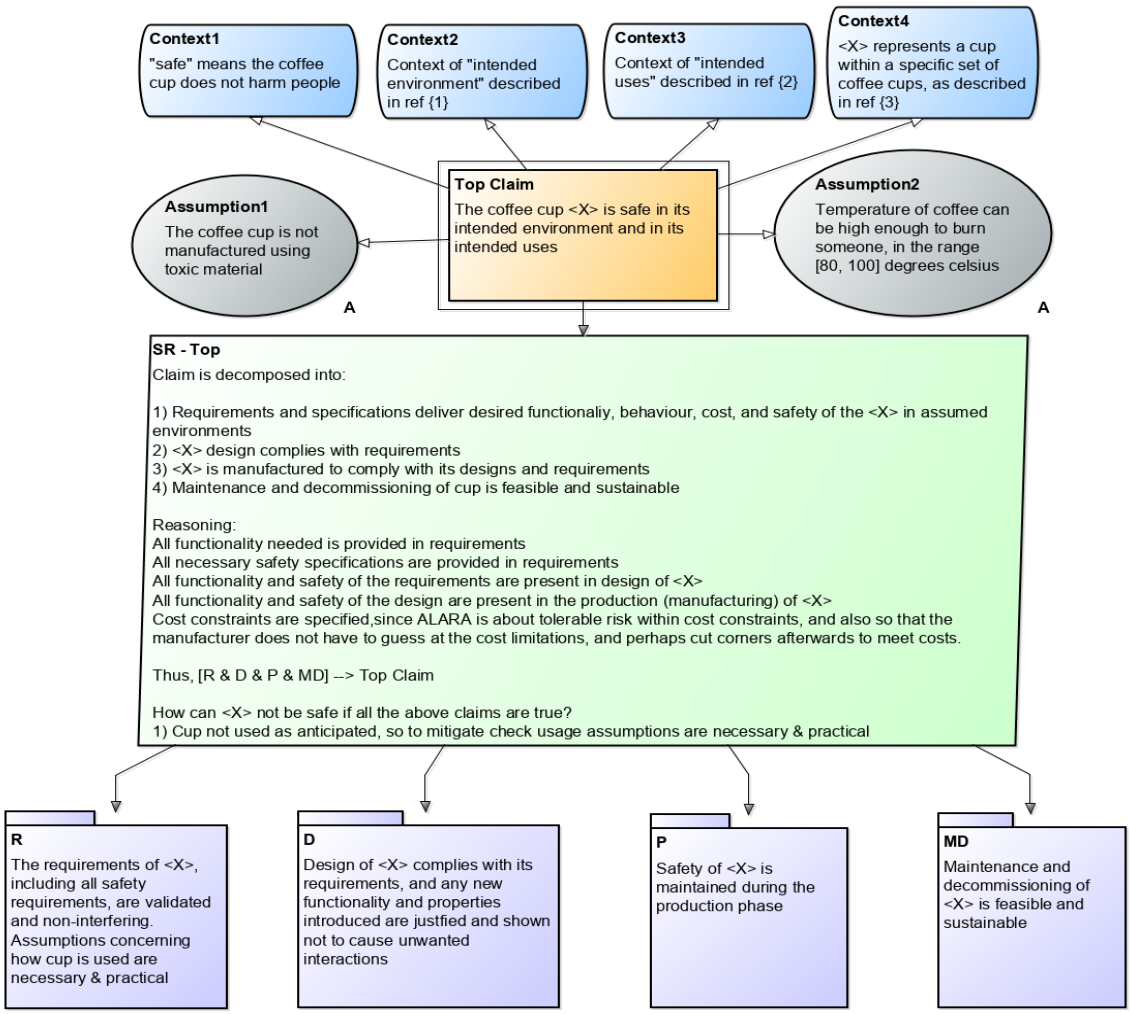

Figure 7: An Example GSN Diagram for Coffee Cup Safety Cases

The Astah GSN XMI file doesn't store modules; thus the user cannot access the modules via Epsilon. Some EOL examples for querying this GSN model are shown in Table 13. The "C!" textual fragment indicates the Astah GSN XMI file. This is needed because Epsilon can run on multiple model files simultaneously, and for running EOL commands on a specific model, you have to use the model's name (C in this case) in the run configuration.

EOL	Result
`C!gsn.SRTop.content.println();`	Claim is decomposed into: …
`C!gsn.topclaim.id.println();`	Top Claim
`C!gsn.assumption1.xmi_id.println();`	_c-Mxh8UxEeq7rp480IS80w
`C!gsn.context1.id = "C1";`	None (Sets element id)
`C!gsn.context.id.println();`	Sequence {Context4, Context3, Context2, C1}

Table 13: Coffee Cup Safety Case GSN Diagram EOL examples

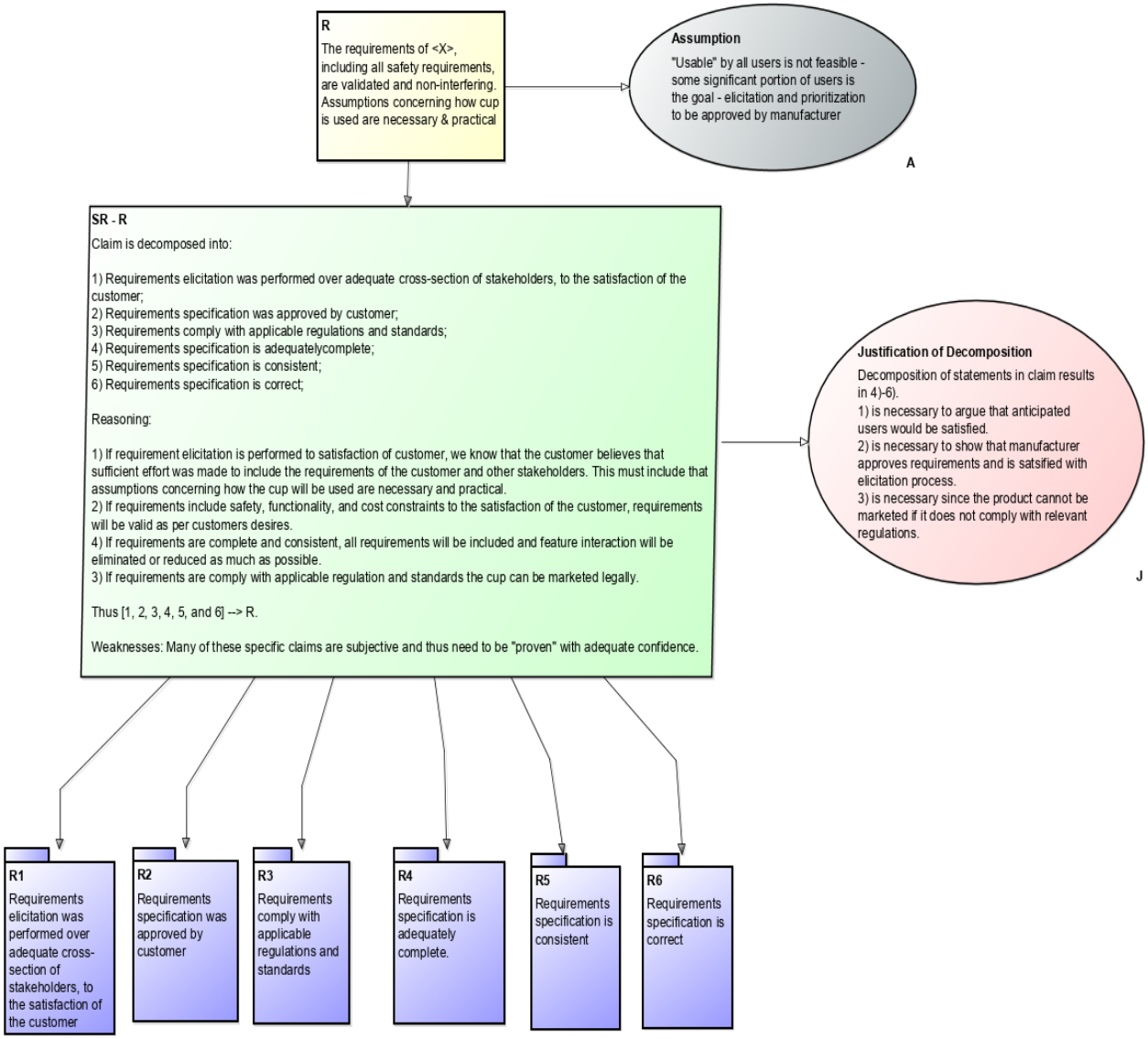

Figure 8: Coffee Cup Safety Cases GSN Diagram R Module

Validation examples for this GSN model are given in Listing 10. The Validation checks the R (Requirement) module's justification element and it should have exactly one source link.

```
context gsn {
   constraint JustificationMustHaveAConnection {
      check: gsn.justificationofdecomposition.source.size() == 1;
      message: "Justification must have exactly 1 connection!"
   }
}
```

Listing 10: Coffee Cup Safety Case GSN Diagram EVL example

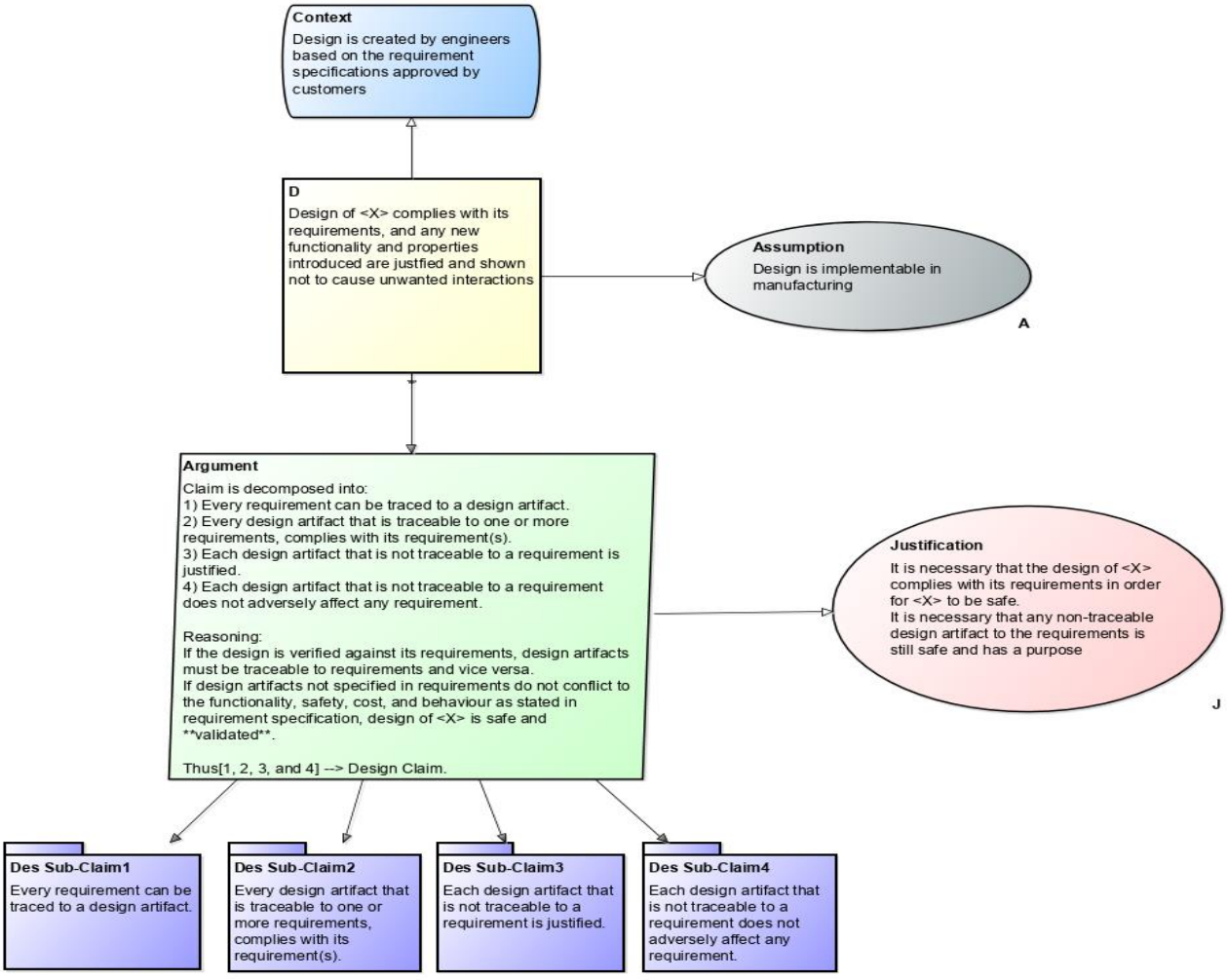

Figure 9: Coffee Cup Safety Case GSN Diagram D Module

A LaTeX table generation EGL script for Coffee Cup Safety Cases GSN diagram is shown in Listing 11. D (Design) module used for this example. It generates a table for each node element. Table's headers contain node IDs and table cells contain node elements' content.

```
[%
var header = gsn.nodes.id.concat("|");
%]
\begin{longtable} {[%=header%]|}
  \hline
  [%for (n in gsn.nodes){%]
  [%=n.content.concat(" & ")%] \\\hline
  [%}%]
\end{longtable}
```

Listing 11: Coffee Cup Safety Case GSN Diagram EGL example

6.3. Requirements Coverage

In this section, an assessment of requirements coverage will be provided, in particular, which of the project requirements has been satisfied will be explained. Reasons for unsatisfied requirements will also be presented, as this may be useful for future research.

As presented earlier, the Astah GSN driver project has 8 main requirements and some of them have additional sub-requirements. Table 14 shows all project requirements status and the list below explains the reasons for the partially satisfied requirements.

No	Requirement Description	Status
1	Users should be able to load Astah GSN models into the Epsilon.	Partially satisfied
2	Users must be able to read/access the GSN models with Epsilon Object Language (EOL).	Partially satisfied
2.1	Users should be able to access the entire GSN model	Satisfied
2.2	Users should be able to access different types of elements	Satisfied
2.2.1	Accessing to all nodes	Satisfied
2.2.2	Accessing to all links	Satisfied
2.2.3	Accessing different types of nodes (e.g. goals, strategies, solutions, …)	Satisfied
2.2.4	Accessing different types of links (e.g. asserted evidence)	Satisfied
2.3	Users should be able to access a specific element in a GSN model	Partially satisfied
2.3.1	Accessing a specific node	Satisfied
2.3.2	Accessing a specific link	Partially satisfied
2.4	Users should be able to access elements' attribute values	Satisfied

2.4.1	Accessing an element's content	Satisfied
2.4.2	Accessing an element's ID	Satisfied
2.4.3	Accessing an element's type	Satisfied
2.4.4	Accessing an element's XMI:ID	Satisfied
2.4.5	Access an element's XSI:TYPE	Satisfied
2.4.6	Access a node element that is the ending point of the specific link	Satisfied
2.4.7	Access a node element that is the starting point of the specific link	Satisfied
3	Users should be able to update GSN models with EOL.	Partially satisfied
3.1	Updating an element's content	Satisfied
3.2	Updating an element's ID	Satisfied
3.3	Updating an element's type	Satisfied
3.4	Updating an element's XMI:ID	Satisfied
3.5	Updating an element's XSI:TYPE	Satisfied
3.6	Updating an element's target link	Satisfied
3.7	Updating an element's source link	Satisfied
4	Users should be able to create new elements and append new elements in a GSN model with EOL.	Partially satisfied
5	Users should be able to delete elements in a GSN model with EOL.	Satisfied
6	Users should be able to validate an Astah GSN model with the Epsilon Validation Language (EVL).	Satisfied
7	Users should be able to transform an Astah GSN model to another model with the Epsilon Transformation Language (ETL).	Satisfied
8	Users should be able to generate code or text from an Astah GSN model with the Epsilon Code Generation Language (EGL).	Satisfied

Table 14: Project Requirements Status

- The first requirement was loading Astah GSN models in Epsilon and it is partially satisfied. Users can load Astah GSN XMI files in Epsilon but they cannot load "AGML" files.
- Requirements #2, #3 and all of their sub-requirements are mostly satisfied. Users can access or update almost all of the given Astah GSN elements. The only exceptions are Goal-to-Strategy and Strategy-to-Goal elements. These two link elements aren't stored in the XMI file so the Astah GSN driver cannot access or update them. However, it is a contribution of this project to have identified this behaviour, which may be something that the Astah organization wants to revisit in the future.
- The fourth requirement is creating and appending new GSN elements and this is mostly satisfied. Users can create a new GSN element and then attach it to the model. However, every GSN elements require a unique *xmi:id* and the Astah GSN driver cannot generate a unique ID value for newly created elements as explained in Section 4.6.
- The last four requirements (#5 - #8) - deleting an element, EVL support, ETL support and EGL support respectively – have all been satisfied and all of them are fully supported by the Astah GSN driver.

7. CONCLUSION

In this project report, the challenges in implementing an Epsilon driver for Astah GSN have been presented. Such a driver bridges the open-source (Eclipse, Epsilon) and proprietary (Astah GSN) worlds and enables rich model management for Astah GSN models. Our vision is that this driver will allow GM engineers to more easily manipulate GSN models using proven and scalable (Epsilon) tools, supporting a rich set of analyses. At the same time, this work is not restricted to use within GM: any user of Astah GSN may benefit from its implementation. It may be that the Epsilon driver could lead to the development of other drivers for assurance case tools (e.g., Medini, ASCE, Certware).

The report described the design options, design updates, implementation difficulties and testing methodologies of the Astah GSN Epsilon driver. The report also demonstrated clearly that the driver can work on Astah GSN XMI files with Epsilon's EOL, EVL, EGL and ETL languages.

7.1. Future Work

This project is a Master of Engineering project and the project time is limited. I conceived of additional features that I simply did not have time to complete. 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, including for maintenance and possible decomposition of very large XMI files. Also, for newly created elements unique *xmi:id* values needed. However, without the necessary information for 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] M. Brambilla, J. Cabot and M. Wimmer, *Model-Driven Software Engineering in Practice.* 2012. pp. 1-23.

[2] D. C. Schmidt, "Model-Driven Engineering", *Computer-IEEE Computer Society*, vol. 39, no. 2, pp. 25-31, 2006.

[3] MIT OpenCourseWare, (2016). Systems Theoretic Process Analysis (STPA). [Online]. Available: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-63j-system-safety-spring-2016/lecture-notes/MIT16_63JS16_LecNotes10.pdf. Accessed: Jul. 10, 2020.

[4] N. G. Leveson and J. Moses, *Engineering a Safer World: Systems Thinking Applied to Safety*. Cambridge: MIT Press, 2012, pp. 1-249. [Online]. Available: https://ebookcentral.proquest.com/lib/mcmu/reader.action?docID=3339365. Accessed: Jul. 10, 2020.

[5] T.P. Kelly, "Arguing Safety: A Systematic Approach to Managing Safety Cases", Ph.D. dissertation, Dept. Comp. Sci., Univ. of York, York, UK, 1999.

[6] T. Ishimatsu, N. G. Leveson, J. Thomas, M. Katahira, Y. Miyamoto, and H. Nakao, "Modeling and Hazard Analysis Using STPA", in *International Association for the Advancement of Space Safety*, AL, USA, May 19-21, 2010.

[7] R. Weaver, J. Fenn and T. Kelly, "A Pragmatic Approach to Reasoning About the Assurance of Safety Arguments", in *Proceedings of the 8th Australian workshop on Safety-critical systems and software*, Australia, 2003, pp. 57-67.

[8] T. Kelly, "A Systematic Approach to Safety Case Management", *SAE transactions*, pp. 257-266, 2004.

[9] T.P. Kelly and J.A. McDermid, "A Systematic Approach to Safety Case Maintenance", in *International Conference on Computer Safety, Reliability, and Security*, Springer, Berlin, Heidelberg, 1999, pp. 13-26.

[10] Epsilon, "Eclipse Epsilon", 2020. [Online]. Available: https://www.eclipse.org/epsilon. Accessed: Jul. 04, 2020.

[11] D. Kolovos, L. Rose, A. García-Domínguez and R. Paige, *The Epsilon Book.* 2010. [Online]. Available: https://www.eclipse.org/epsilon/doc/book/EpsilonBook.pdf. Accessed: Jul. 10, 2020.

[12] D.S. Kolovos, R.F. Paige and F.A. Polack, "The Epsilon Object Language (EOL)", in *European Conference on Model-Driven Architecture-Foundations and Applications*, Springer, Berlin, Heidelberg, 2006, pp. 128-142.

[13] A. Zolotas, H.H. Rodriguez, S. Hutchesson, B.S. Pina, A. Grigg, M. Li, D.S. Kolovos and R.F. Paige, "Bridging Proprietary Modelling and Open-Source Model Management Tools: the Case of PTC Integrity Modeller and Epsilon". *Software and Systems Modelling*, vol. 19, no. 1, pp. 17-38, 2020.

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

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

[16] Jonathan Hedley, “Jsoup: Java HTML Parser”, 2020. [Online]. Available: https://jsoup.org. Accessed: Jul 24, 2020.

[17] Java SE Documentation, “Document Object Model (DOM) Library Documentation”, 2020. [Online]. Avaliable: https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/org/w3c/dom/package-summary.html. Accessed: Jul 30, 2020.

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

[19] T. Chowdhury, A. Wassyng, R. F. Paige and M. Lawford, "Criteria to Systematically Evaluate (Safety) Assurance Cases," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 2019, pp. 380-390, doi: 10.1109/ISSRE.2019.00045.