Alloy Specification Generator

Translate drag-and-drop components into Alloy specifications

Abbie Jarratt
Computer Science
The University of Nebraska
Lincoln NE USA
abbie.jarratt@gmail.com

Trent Hoppe
Computer Science
The University of Nebraska
Lincoln NE USA
hoppetrent06@gmail.com

Jacob Miller
Computer Science
The University of Nebraska
Lincoln NE USA
16jamill16@gmail.com

Introduction

There exists a variety of software languages that aim to model different architectural system. However, there is currently no software that is able to translate a common architectural diagram into the code required for many advanced software systems. To combat this issue, the Alloy Specification Generator (ASG), was created. ASG allows a user to design a diagram and then export the diagram into Alloy specifications.

The generator is constructed of a user interface that can transfer the data to a component that contains the logic to translate a diagram into an Alloy specification. With this tool, a user will be able to drag-and-drop components, connectors, roles, and ports in an intuitive user interface. Each of these drag-and-drop components will have a label and their attachments will be visible within the canvas. Based on this diagram, ASG will convert this diagram into a set of Alloy instructions.

This will allow architects or other users to quickly and easily see alternate configurations based on the known constraints. This will enable users to test different configurations or make updates more quickly than if they were to write the Alloy instructions themselves. ASG will save time and effort for its various users.

KEYWORDS

alloy, code translator, specifications, model constraints, system architecture, architecture diagramming

1 Background

This section will provide an overview of the different technologies used within the project. It will discuss, Alloy, Fabric.js, and C# .NET web application solutions.

1.1 Alloy

Alloy is an open source programming language that assists users with software and architecture modeling. The language is relatively young, being founded in 1997. Its developers wanted Alloy to be a declarative specification language that would enable a variety of users to model systems. These systems can range from describing the structure of complex security networks to simple pipe and filter systems. When a system has been created, it is regarded as an Alloy model. The Alloy model is an inclusive collection of "constraints that [describe] (implicitly) a set of structures" [2]. These constraints can be represented through signatures, facts, predicates, functions and assertions. A description of each follows:

Signatures: A signature declaration defines a new component or element of a system. Within the declaration body, a use may add relationships or various constraints on a system. For example:

```
sig FSObject {}

sig Dir {contents: set FSObject} [2]
```

In this case, an FSObject and a Dir were created. There is also a relationship between the two stating that multiple FSObjects are the contents of a Dir object.

Facts: A fact is a statement that will be true throughout the system. These are typically used to set constraints on relationships within a model. For example,

```
fact {a   ll d: Dir, o: d.contents | o.parent = d} [2]
```

This translates to "Given any directory, d, and any (FSObject) o, in d's contents, d must be the parent of o" [3].

Predicates: The use of a predicate allows users to introduce new constraints and have Alloy produce instances that

satisfy these new constraints. Predicates are commonly used as a quick way to visualize the current state of a system. An example is included below:

```
pred show () {}
```

Functions: Similar to other programming languages, functions in Alloy use expressions to return a result. A commonly used Alloy function is the "lookup" function which allows user to return all values associated with a constraint.

Assertions: Within Alloy, assertions are used to test how the program is supposed to behave. These are especially important for testing and reasoning.

Once an Alloy model has been created, it can be used to analyze different possible configurations for a system. This is done through the Alloy Analyzer that uses the given constraints to solve for possible solutions. This can help discover security vulnerabilities or topologies of a switching network. Each solution is presented in a graphical interface which allows users to quickly visualize different options.

1.2 Fabric.js

Fabric.js is a simple JavaScript library released in 2016 [4]. It allows developers to easily manipulate HTML5 canvases. An HTML5 canvas is used to draw and manipulate graphics on a web page. Thus, the Fabric.js library simplifies the code and work necessary to create graphics and images. It has been pre-populated with simple geometric shapes and also complex shapes. It also allows the user to easily update colors, sizes or add text to a shape. One major benefit of the library is its built-in dynamic behavior. Shapes can be manipulated or moved through the library with no additional work from the developer [4].

1.3 C# .NET

The C# language was developed by Microsoft specifically to work within the .NET framework. It is an object-oriented language that is type-safe by design. Despite being built for the .NET framework, it is a general-purpose programming language that has been used to create a multitude of solutions [6]. The first version, C# 1.0 was released in 2002 followed by a range of updates until it reached C# version 7.3 in 2018 [5]. These updates led to the language becoming a standard by ECMA and ISO. Solidifying its flexibility and robust solution set.

Figure 1: **Microsoft C# and .NET Framework Logo**

The .NET (pronounced dot net) framework was developed to enable developers to execute software in a software environment as opposed to a hardware environment. This means that applications produced using the .NET framework should run the same regardless of machine type. The framework provides security, memory management, and exception handling for the developer. It has also led to the development of the Framework Class Library (FCL) which allows different programming languages to utilize code written in other languages. Today, .NET is an open-source solution that is used by over six million developers [7]. When combined with C#, it has become a stable of web development for its ability to create powerful, robust web applications.

2 Technical Overview

The first portion of development, was devoted to system architecture and design. At this time, the development team chose to use C# .NET and Fabric.js to build their solution. This technology stack was decided due to the simplicity of implementation and familiarity with languages. The team created the prescriptive architecture outlined in Figure 2.

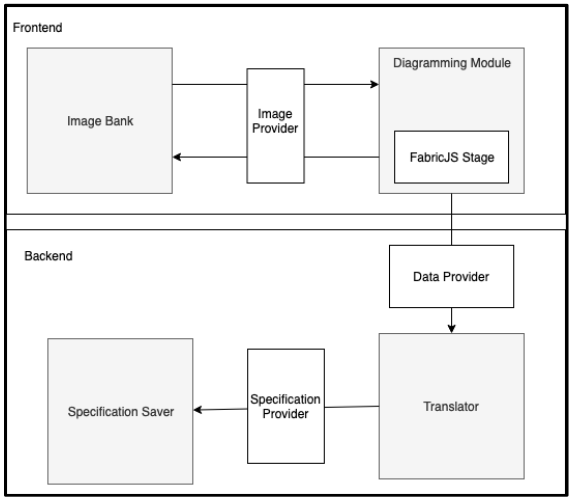

Figure 2: **Prescriptive architecture of the ASG**

In this design, there are four major components, Image Bank, Diagramming Module, Translator, and Specification Saver. The Image Bank was responsible for housing different templates for users to use as a basic design. Then, the Diagramming Module would provide users with the ability to move and manipulate the diagram. A user could create a new diagram and select a Client and Server template which would be pulled from the Image Bank using the Image Provider. Within this system, Fabric.js would only exist within the Diagramming Module to manipulate the canvas. This logic would comprise the entirety of the frontend system.

Then, to send information from the frontend to the backend the data provider would be used. This will be transferred using JSON to the Translator component. Within that component, the translation from JSON to Alloy would take place. Finally, it would be sent to the Specification Saver by way of the Specification Provider to be saved to the client side. This was the architecture determined prior to beginning development.

After implementing the system, the development team recreated the system's architecture diagram. That diagram is displayed in figure 3 below.

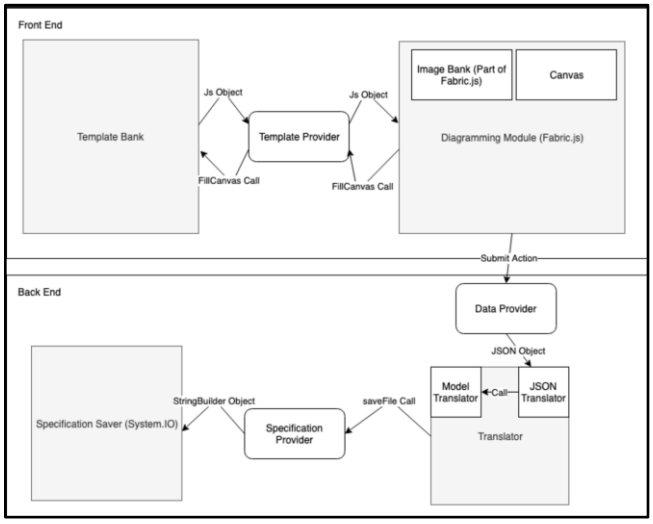

Figure 3: **Descriptive architecture of the ASG**

During development, the team realized that the Image Bank had been misnamed. Thus, this was correctly titled Template Bank in the implementation. In the descriptive diagram, most interactions remained consistent. The differences between the diagrams are a result of detail and a better understanding of development tools. For example, the development team was able to label the type of data being transformed and even specific function calls within components.

3 Technical Details

3.1 Template Bank

The Template Bank is located in the Scripts > Source > Models > Templates.js file. There are currently two templates, a Pipe and Filter, and Client and Server template. Each of these templates has a variety of components, connectors, roles, ports, and interactions. These are all saved using JSON which is parsed and displayed by the Diagramming Module.

```
// Pipe and Filter Templates
const PIPE_AND_FILTER = {
    type: "PipeAndFilter",
    components: [
        {
            id: 1,
            name: "DataSource"
        }, {
            id: 2,
            name: "DataSink"
        }
    ],
    connectors: [
        {
            id: 3,
            name: "Pipe1"
        }
    ],
```

Figure 4: **A code snippet showing the JSON behind the Pipe and Filter template**

This component will allow for the future developers to easily add and replace templates. The templates can be switched out or edited with ease. This also enables stand-alone components that can easily accept or send JSON to and from other components. This will help with reuse and redesign for future enhancements.

3.2 Diagramming Module

The Diagramming module is comprised of javascript code and the Fabric.js library. Within the project solution, this code can be found in the Script > Source > Diagrammer.js file. This file contains all of the logic for creating a new diagram, adding and editing components or connectors, and manipulating their connections.

```
(function () {
    initializeDiagrammer();
    //initializeNewDiagram("Pipe and Filter");

    $("#new-diagram-form").on("submit", function (event) {
        event.preventDefault();
        clearArchitectureElements();
        initializeNewDiagram();
        $("#new-diagram-form").modal("hide");
        return;
    });
```

Figure 5: **A code snippet showing the initialize diagrammer function which renders a new diagram for the user**

The purpose of this component is to encapsulate the logic for diagram creation. This file is responsible for displaying any updates to the diagram and also condensing it into a JSON object. This allowed for separation between the diagramming module and the translator, meaning each component can be updated without affecting the other.

3.3 Translator

The translation component was written entirely in C#. All of the logic can be found in the Translator folder within the project solution. Currently, there are two architecture styles implemented, Client and Server, and Pipe and Filter. Thus, within the Translator folder there is a translator and creator for each architectural style.

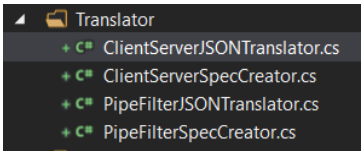

Figure 6: **File contents of the Translator folder**

The translator is responsible for parsing the JSON object which will be received from the frontend. Then, the parsed object is sent to the creator where the Alloy specifications are generated.

```
public class ClientServerJSONTranslator : IClientServerJSONTranslator
{
    public ClientServerSystemModel CreateModel(String json)
    {
        var details = JsonConvert.DeserializeObject<JsonDetails>(json);

        ClientServerSystemModel system = new ClientServerSystemModel();
        system.Name = "ClientServerSystemExample";

        // For each Connector
        foreach (Connector c in details.connectors)
        {
            CnsConnector newConnector = new CnsConnector();
            newConnector.Name = c.name;
```

Figure 7: **A code snippet from the ClientServer-JSONTranslator.cs file located within the Translator folder**

The translator component allows all Alloy specification logic to be encapsulated. This means that maintenance or additional features can easily be added within the component without interrupting other component's roles.

3.4 Specification Saver

The logic responsible for downloading the Alloy specifications to the client's computer is located within the Controller folder. The component utilizes a Response object to save the document on the client side.

```
var contentType = "text/plain";
var fileName = "alloySpecification.als";
var header = "attachment;Filename=" + fileName;

Response.Clear();
Response.Buffer = true;
Response.ContentType = contentType;
Response.AppendHeader("Content-Disposition", header);

Response.Write(sb.ToString());
Response.Flush();
```

Figure 8: **A code snippet displaying the file saving logic**

This is an extremely important component within the system. It is responsible for downloading the Alloy specifications regardless of the environment. As such, the development team opted to split out it's logic from the translator. This way, any updates can be controlled and minimized while retaining the integrity of other components.

4 Demo

Once the project is running, the user can view an html page by visiting localhost:51023. (Note: please refer to the project read.me if there are any issues with executing the solution) Here, the user will be able to create a new diagram, edit and update a diagram or generate an Alloy specification.

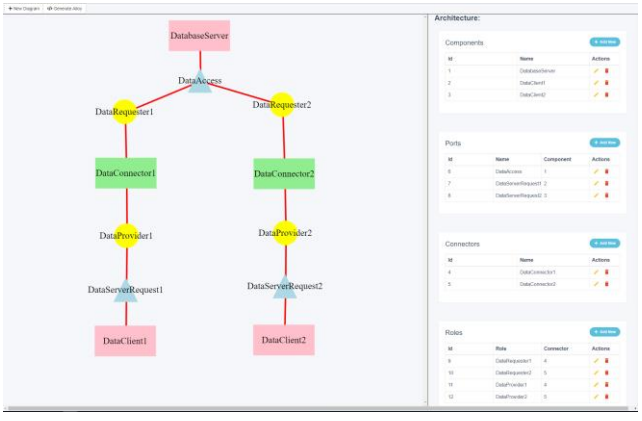

Figure 9: **The main page of the ASG application**

After selecting a diagram, the user can rename components and connectors along the right-side panel. This panel also controls the interactions or connections between components. After making the final additions to their diagram, the user can click the “Generate Alloy” button along the top bar. This will send a JSON object to the translator which will create an Alloy file. The user will then be asked to enter a file name, which will appear as a traditional download through the Chrome browser. Note, the ASG is currently optimized for the Chrome browser.

alloy-specification

```
module CnsExample

open library/Cns_simplified as Cns

one sig ClientServerSystemExample extends System{}

one sig DataConnector1 extends CnsConnector{}{
    roles = DataRequester1 + DataProvider1
}

one sig DataConnector2 extends CnsConnector{}{
    roles = DataRequester2 + DataProvider2
}

one sig DataClient1 extends Client{}{
    ports = DataServerRequest1
}

one sig DataClient2 extends Client{}{
    ports = DataServerRequest2
}

one sig DataProvider1 extends Provider{}

one sig DataRequester1 extends Requester{}

one sig DataProvider2 extends Provider{}

one sig DataRequester2 extends Requester{}

one sig DataServerRequest1 extends ServerRequest{}

one sig DataServerRequest2 extends ServerRequest{}

fact Correct_Attachments {
ClientServerSystemExample.attachments =
DataRequester1->DataAccess
+ DataRequester2->DataAccess
+ DataProvider1->DataServerRequest1
+ DataProvider2->DataServerRequest2
}

pred show {
    one System
}

run show
```

Figure 10: **The Alloy specification created through ASG**

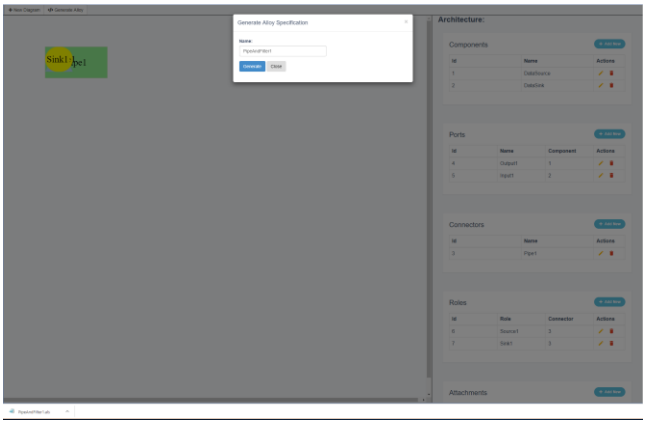

Figure 11: **Screen capture of Alloy model downloading to client desktop**

At this point, the user can upload the file through the Alloy application to receive the configurations based on the current constraints outlined.

5 Evaluation

This project will enable users to build Alloy specifications from a GUI system. The ASG reduces the strain on architects to learn and fully understand Alloy syntax. Also, it will assist new Alloy programmers with learning the language while breaking down some of the confusing code blocks. This will effectively shorten the learning curve associated with building Alloy models. Thus, more programmers will be able to use the Alloy system to design and architect their systems. This will introduce consistency and the ability to revise an architecture before settling on a final version.

The main contribution for architects is the ability to easily test different configurations of a software system. Now, programmers will be able to add and remove components from their diagrams and quickly check their updated Alloy model. This will save precious time for the developers while still retaining the same valuable results. Users will still be able to test an architecture before putting it in place and also visualize different constraints on that system.

ASG can also benefit developers who are unfamiliar with architecture design. The solution will have base templates for Client and Server and Pipe and Filter which will provide users with a jumping off point for system design.

Overall, the ASG will allow architects to develop accurate Alloy specifications quickly, easily and effortlessly. In turn, this will lead to more thoughtfully developed systems and fewer design errors in the prescriptive architecture.

6 Division of Work

The work for this project was divided among three developers, Trent Hoppe, Abbie Jarratt, and Jacob Miller. The prescriptive architecture was determined during an all-group meeting. The division of work for the project took place at this same time. The team determined the following breakdown of work to fulfill the requirements:

- Translation Component: Jacob Miller
- Saving Component: Abbie Jarratt
- Specification Provider: Abbie Jarratt
- Diagramming Module: Trent Hoppe
- Data Provider: Trent Hoppe
- Bug Fixes and Refactoring: Jacob Miller

The project was divided based on previous experience and interest. For example, Hoppe is familiar with javascript and the Fabric.js library. Thus, the team determined he would be in charge of the diagramming module. The rest of the schedule was determined using the same methodology.

7 Future Work

During development, the team determined a variety of additions to be added to the second version of ASG. These are advancements that would increase the usability and robustness of the solution. Both additions are described in detail below.

7.1 Additional Architecture Templates

Currently, the ASG can create Pipe and Filter, and Client and Server architectures. These two architectures were determined by the development team based on their stability and wide variety of use cases. However, there are still multiple architectural styles that remain to be implemented. These include by are not limited to Blackboard, Event-driven, Layered, Peer-to-Peer, and Event-bus. The implementation of these templates would mimic those of the Pipe and Filter, and Client and Server.

By implementing these additional architectural styles, this will broaden the use cases in which ASG can be applied. It will also assist new architects by introducing them to multiple architectural styles. This will allow them to quickly switch between model designs and see the desired output. The team believes this will provide value and should be added to the version two requirements.

7.2 Advanced Facts

One feature within Alloy that is not currently available through ASG is the ability to enter facts. As discussed in section 1.1, facts allow developers to define constraints on a system. These can range in complexity and are commonly used to define relationships between objects. The current version of ASG does not support the addition of facts. Thus, the development team has identified this feature to be added to the second version of ASG.

The addition of facts into the system will introduce complexity and require the user to have a basic understanding of their purpose. The team would suggest implementing a ‘Facts’ tab within the current user interface. Within this tab, a user could add a new fact which would populate drop-down inputs based on the architecture diagram. Then, if an update was made to the architecture diagram, the drop-down options would update dynamically.

This would still require some knowledge of facts on the user end, but would immensely reduce complexity. This will also allow ASG to be applicable to a wider variety of systems, further expanding its solution set. The addition of facts would shorten the time required for small updates and maintenance and improve the usefulness of ASG.

8 Conclusion

The ASG was created to save time and effort for software architects. The solution allows users to drag-and-drop various components, connectors, ports, and roles in order to design new systems. Then, using the diagram produced, ASG will create an Alloy model. Given an Alloy model, the user can then visualize different configurations of their design. This allows them to utilize the capabilities provided by Alloy without knowing the Alloy language itself. The complexity is encapsulated by the ASG which simply takes a diagram as an input and produced the desired Alloy model. This system simplifies the current process of building Alloy specifications while retaining the value provided by Alloy.

This project could provide immense value for architects in the future. This value would be amplified through the additions of advanced features mentioned above. However, in its current state, ASG will simplify the process of building Alloy specifications.

REFERENCES

[1] Kyriakos Anastasakis. 2009. *A model driven approach for the automated analysis of UML class diagrams*

[2] Alloy Analyzer website [online]. [cited April 2008]. Available from: URL: http: //alloy.mit.edu.

[3] UML2Alloy website [online]. [cited April 2008]. Available from: URL: http://www. cs.bham.ac.uk/~bxb/UML2Alloy/index.php

[4] "Developer Docs." *FabricJS*, fabricjs.com/docs/.

[5] "History of C# Programming Language." *C-SharpCorner*, c-sharpcorner.com/blogs/history-of-c-sharp-programming-language.

[6] "C#." *Microsoft*, docs.microsoft.com/en-us/dotnet/csharp/.

[7] ".NET" *Microsoft*, docs.microsoft.com/en-us/dotnet/.