Security Assurance Case Design Tool

Client/Advisor: Dr. Lotfi Ben-Othmane
John Koehn
Brandon Huegli
Jordan Lawrence

Team Dec1718

1. Revised Design

This section goes over the goal of the project, the requirements and the design.

1.1 Project Goal

The main objectives for this project were to develop an Eclipse plugin based application for the creation of assurance case diagrams, and implementing features to assist in verifying the continued validity of the claims within that case. These features took the form of allowing the application to use Git integration to check if a diagram element's entered "entry point" function was affected by the most recent Git commit. This software is being developed for Dr. Othmane at Iowa State University, with the hope of helping further his research related to assur ance cases. See fig. 1.1.1 for an example of what a security assurance case diagram might look like.

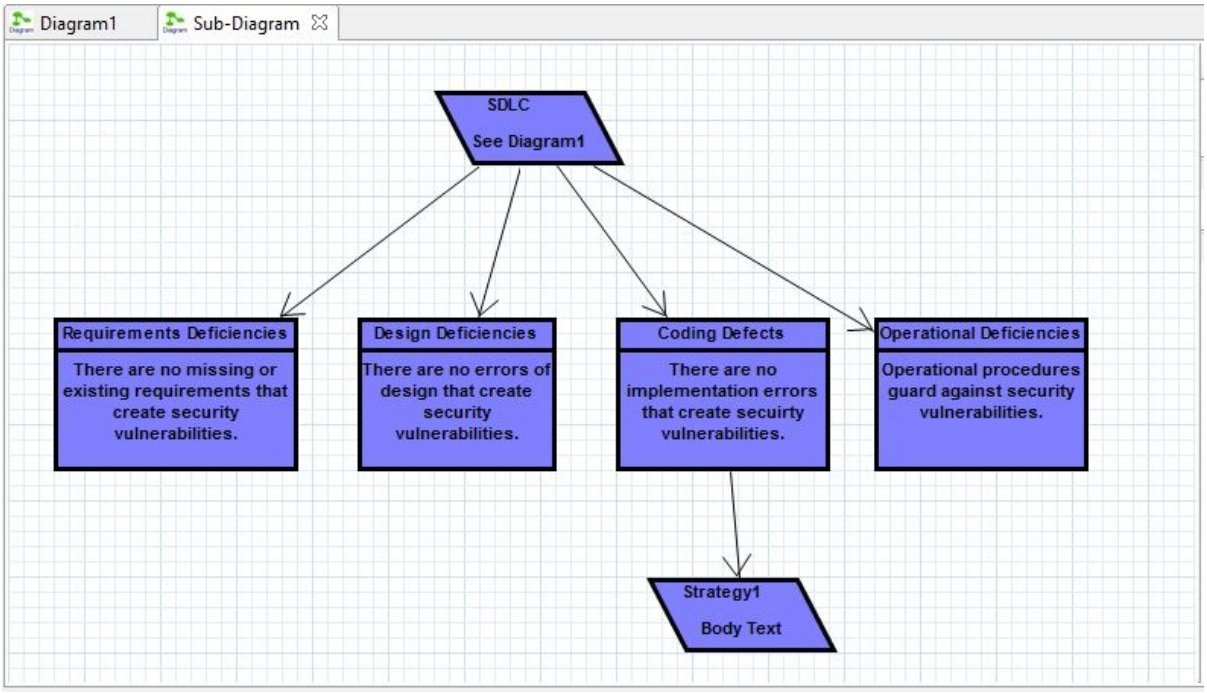


Fig. 1.1.1 Example security assurance case

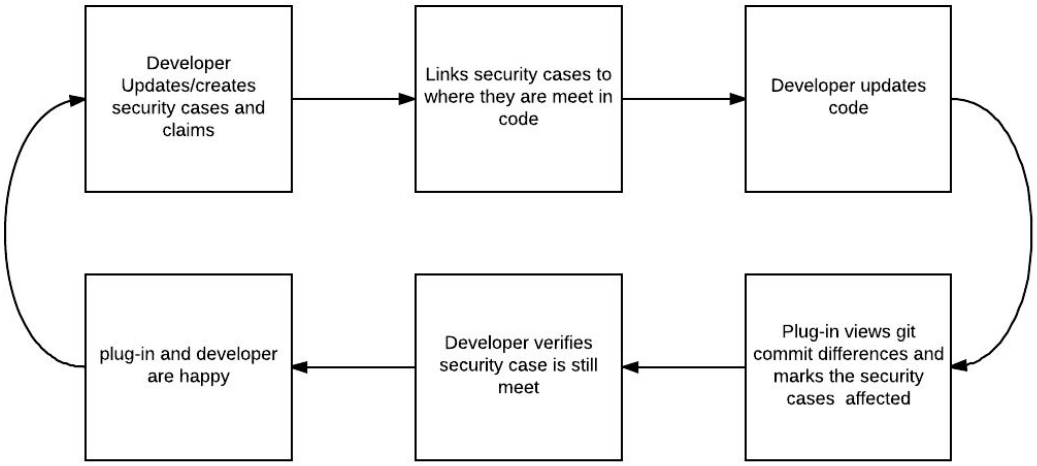


Fig. 1.1.2 General application flow

In the Fall semester, we primarily worked on completing the user interface functionality of the software, including the diagram elements and the necessary features of that. This semester, we worked to add the ability to generate and parse a function call graph, and use a Git change log to find modified functions and mark diagram elements that contained an affected function as its entry point function. See fig. 1.1.2 for the general concept flow of the project.The design and implementation of these two aspects are discussed further in depth below. See Appendix I for reference images and text describing the operation of the software.

1.2 Requirements

While our project was developed in a somewhat iterative manner with input from Dr. Othmane guiding our sprint planning, we developed some basic key functional and nonfunctional requirements that would need to be met for the project to be successful.

1.2.1 Functional

1. The diagram editor shall contain all the necessary elements to model a security assurance claim.
2. The diagram editor shall allow elements to be moved, resized, added, removed, etc. freely.
3. A diagram element shall be able to be linked to another, separate diagram file.

4. Diagram elements shall contain a field for entering descriptive data, and another for entering code entry points connected to that element.

5. When an entry point function, or any function utilized directly or indirectly by that function changes, the diagram elements having that entry point will be visually identified.

6. Changed functions will be retrieved from a connected Git repository.

7. Child function calls will be identified by generating a traversing a call graph using the Wala library.

8. The call graph functionality shall support Javascript embedded in an html file.

9. Diagram files, and all connected data should be able to be saved and loaded.

1.2.2 Non Functional

1. The application's interfaces should be visually intuitive to an end user.

2. The application must be implemented as an Eclipse plugin.

3. The application should be responsive to under a second, except for the call graph generation, which necessarily takes longer.

1.3 Component Design

The application's functionality can be essentially divided into two main components; the user interface, encompassing the need to create and edit diagrams, and the integrated ability to determine the methods/functionality affected by a project's associated Git change log, which is visually represented into the diagram. The design originally gonna have test cases to verify security claims would be meet. However, due to the complexity of incorporating the feature, we had to remove it from our design.

1.3.1 Diagram Editor

The user interface was implemented using the in-development Eclipse framework Graphiti, which is designed to meet use cases similar to our application; the development of Eclipse plugins featuring graph/diagram editors. Graphiti provides some base functionality in terms of creating and saving the actual diagram files, and otherwise provides a somewhat rigid structure for implementing the details of the editor. At a simplistic level, Graphiti requires the implementation of specific interfaces for diagram elements or functionality, such as the shapes and connecting arrows, or the ability to resize elements. These classes are then "registered" or included in what Graphiti calls "providers" which allow access to those items within a diagram file.

1.3.2 Git and Code Integration Functionality

This functionality allows the diagram to be connected in some ways to the code within the Eclipse project, and the project's Git repository in relation to the assurance case being modeled in the diagram. A "claim" element within the diagram can define an entry point, or which functions are related to that claim's validity. The Git change log is then retrieved for the last commit, and if that entry point, or any functionality used by that entry point is modified the claim is visually marked invalid for the user to then revalidate.

This ability was achieved using Wala to generate a call graph from changed functions retrieved from Git through JGit. When the user selects "generate callgraph" from the menu, a call graph is generated for each modified file, and the graph is traversed to check if any of the changed methods or their predecessors have been modified, and are a defined entry point for any element of the diagram.

2. Testing

Below is a description of our testing process and the results of that methodology.

2.1 Process

We worked on weekly to bi-weekly sprints based on the scope of the planned changes. Every week we would report and demonstrate our progress to Dr. Othmane, where he would then provide feedback and guidance on additions, changes, or fixes. We would incorporate new feedback into our sprints to make sure our vision and Dr. Othmane’s vision of the product was the same. In this way, our testing process mirrored the development itself as a continually evolving process.

2.2 Results

While software is often tested with a more explicit testing phase and with test cases, the iterative process we used was suited to our development. Going into this project we had little knowledge of the project's domains, like security assurance cases and eclipse plugin development. Our iterative process allowed us to get quicker feedback on our progress, and ensure the product we were developing was inline with Dr. Othmane's needs.

3. Related Products

The main functionality of our project focuses around the diagram editor, and interactions involving diagram elements. In the beginning design and planning stages for the project, we looked at a range of technologies and frameworks before deciding on Graphiti to develop that editor. The rest of the project is mainly plain Java, and so this decision was one of the main design choices we faced.

The first candidate library we found was JGraphx, which is built on top of Java Swing. This means we could've began developing with it much faster, as we all had familiarity with Swing, and its documentation is pretty good. However, JGraphx had some downsides as well; we felt Java Swing was not the strongest approach to the very interactive, dynamic GUI demanded by our project. The library also had a very dated aesthetic; while this isn't necessarily very important, when compared to our other options we decided against using JGraphx.

The next framework we considered turned out to be what we used in the end, Eclipse Graphiti. Graphiti was created for making eclipse plug ins with charts/diagrams/graphs, which would fit into our design goals well. It also had an acceptable, if somewhat lacking set of documentation, and had a more up to date aesthetic appearance. The only real concern with using this was our lack of experience with it, and the fact that it is still in early development, not even considered a full release. However, we felt this was our best option.

Next we considered JGrapht. Like JGraphX, this framework was built on top of Java Swing, so it also came with the same drawbacks and advantages of that. Additionally, the design goals of JGrapht did not as closely align with our project; we felt its intention was for a different style of diagrams than we needed. This meant it was a possible workable option, but far from ideal and not comparable to the better options available.

Finally, we looked at Graphviz. This appeared to be able to generate the style of diagrams we needed, but had very poor available information, and did not appear to be built of Java, or Eclipse. Like JGrapht, this was an option we found, but did not really consider a serious contender compared to Graphiti or JGraphx.

Summary Chart

JGraphx*[1]	Would be able to meet the requirements of the project. Would be easy to begin development with.	Built on Java Swing. Not eclipse specific. Dated aesthetic.	We decided that while this framework was a candidate as it met our requirements, it was not the best option available.
Eclipse Graphiti*[2]	Built for making eclipse plug ins. Aesthetically the	Team lacks experience with plug in development.	The team has chosen Graphiti for development of the project.

	most pleasing. Acceptable documentation and tutorials.		
JGrapht*[3]	Would've been likely able to meet requirements in most ways.	Built on Java Swing. Not intended for the exact style of diagram necessary.	This framework was not considered a candidate, as it did not match our needs as closely as other options.
Graphviz*[4]	Able to draw diagrams needed.	Poor available information. Doesn't appear to be built for Java or Eclipse.	As with JGrapht, this library was not considered a candidate.

4. Related Literature

At the beginning of this project, none of the team had any knowledge of assurance cases, and obviously in order to design a tool based around the creation and use of such cases, we did some research into them. Our research turned up three academic sources that discuss assurance cases in general, and describe their structure. These served as a basis for our understanding of the overall goal of our project, and shaped the design elements within our diagram editor user interface. We also found one existing project that has some similar goals as our project, though it was no longer maintained; however, the documentation and discussion on the software's site was also useful in helping us understand our project.

The first document is a paper from Carnegie Mellon University by John Goodenough, Howard Lipson, and Charles Weinstock. This paper for the most part just described what a security assurance case is, what they are used for, and their structure. It was very useful to us when designing our user interface, as it provides a list of necessary elements, and differentiates the separate elements in an assurance case diagram. It doesn't really discuss assurance cases designed within software, or tools to create/verify them, but is otherwise the theoretical basis for our work.

The next document is a conference paper by Cristophe Ponsard, Gautier Dallons, and Philippe Massonet. This paper is a more in depth look at assurance cases, and provides a case study to discuss concerns on designing and then continually verifying cases and software changes and evolves. This paper also provided some valuable information on what we needed to consider when working on our project.

The final paper we looked at was another from Carnegie Mellon, and was essentially a summary of the other paper, offering a description of assurance cases in general and their structure, but it also discusses other notations for creating them. This was useful to us as a lighter, quick reference for certain design elements, but is otherwise less comprehensive than the others, and didn't offer much new information.

Finally, we looked at some documentation and papers linked with NASA's "CertWare" software. While we were not able to actually use their software, the website it is hosted on is still up, and offers a selection of papers discussing their design, and assurance cases in general. It offered us a somewhat less purely academic source, and gave us some idea of the software that already exists within the domain of our project.

References

Goodenough, John, Lipson, Howard, and Weinstock, Charles. “Arguing Security - Creating Security Assurance Cases.” Carnegie Mellon University, 4 November 2017. https://www.us-cert.gov/bsi/articles/knowledge/assurance-cases/arguing-security-creating-security-assurance-cases. Accessed Feb. 2017

Ponsard C., Dallons G., Massonet P. (2016) Goal-Oriented Co-Engineering of Security and Safety Requirements in Cyber-Physical Systems. In: Skavhaug A., Guiochet J., Schoitsch E., Bitsch F. (eds) Computer Safety, Reliability, and Security. SAFECOMP 2016. Lecture Notes in Computer Science, vol 9923. Springer, Cham

Software Engineering Institute. Carnegie Mellon University, 2017, http://www.sei.cmu.edu/dependability/tools/assurancecase/. Accessed Feb. 2017.

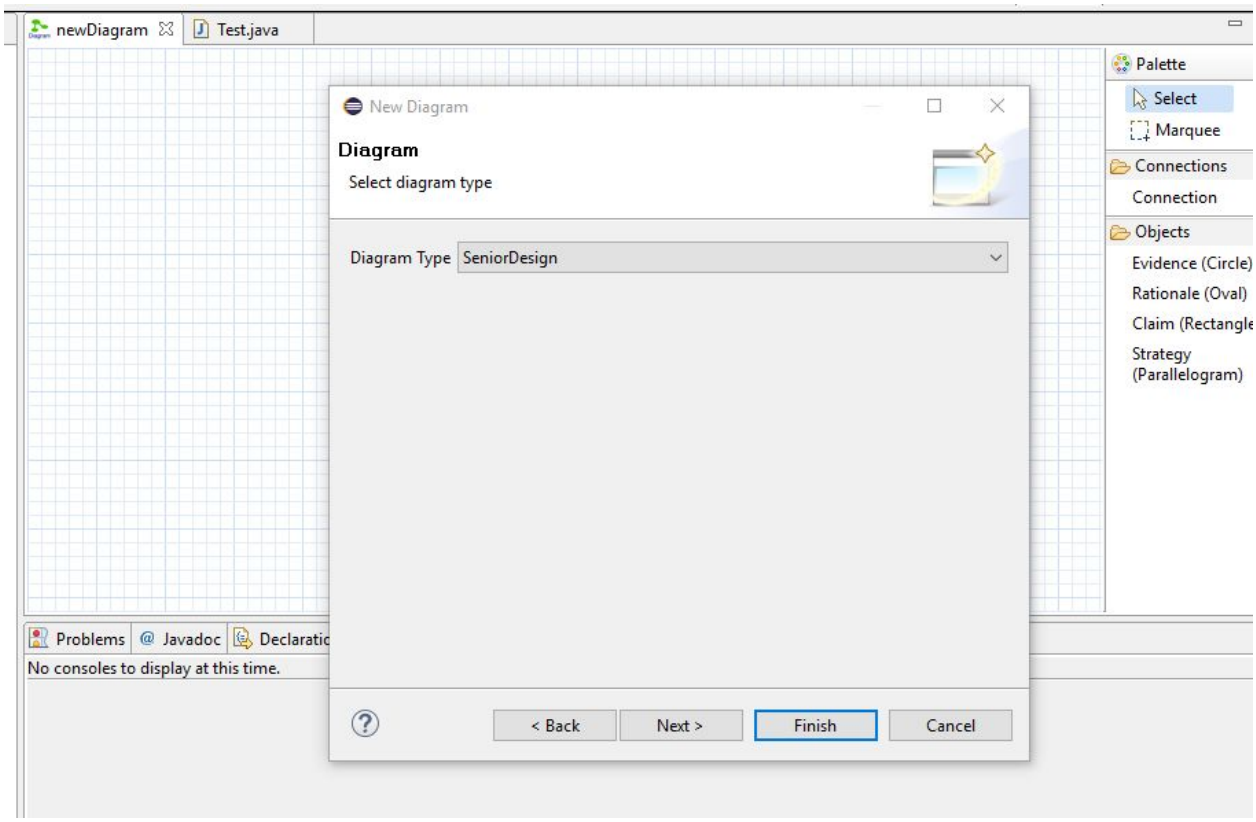
Certware. Nasa, 2012. https://nasa.github.io/CertWare/. Accessed Feb. 2017

Appendix I - Operation of Software

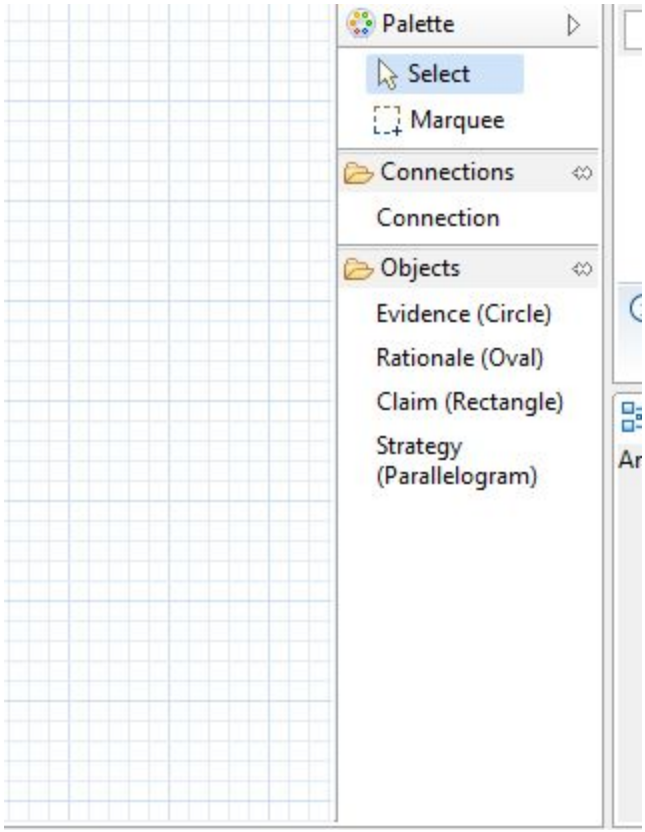
1. Install the plugin to an Eclipse installation.

2. Select File -> New -> Other -> Graphiti Example Diagram -> Senior Design

    A new diagram file will be created in the current/selected directory.

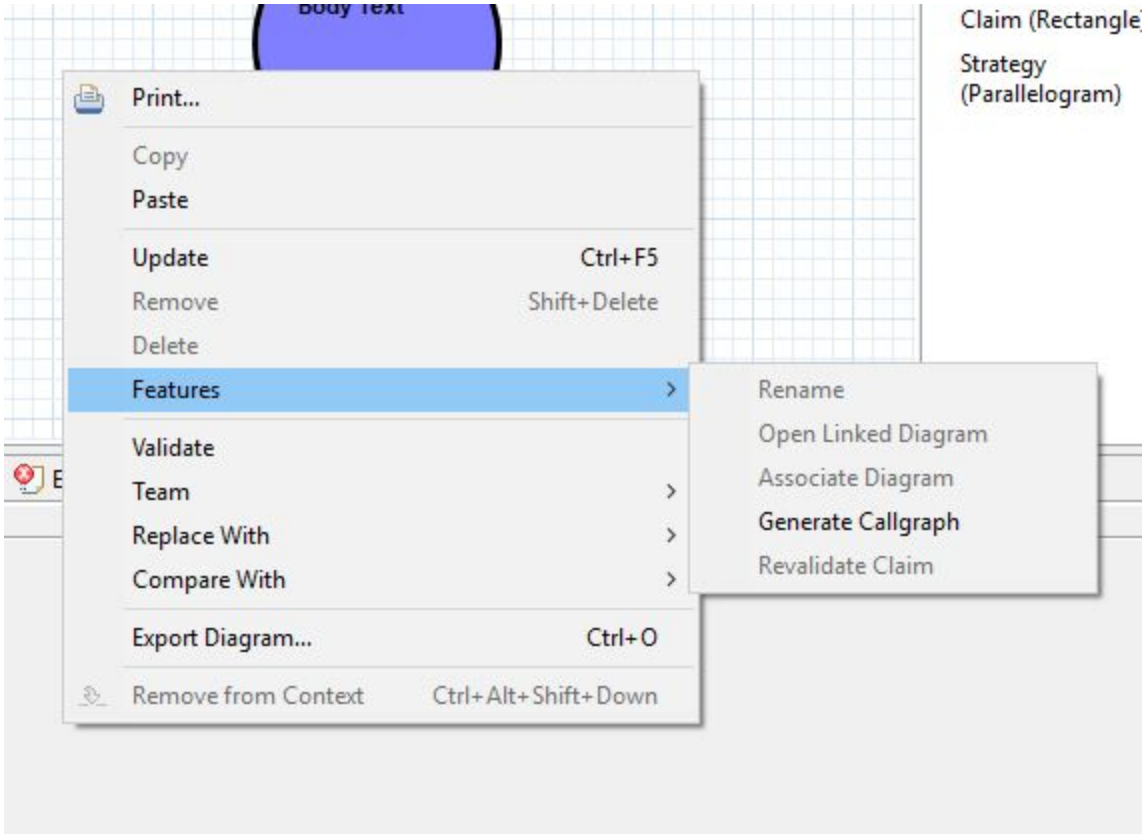


3. Open the diagram file in the Eclipse editor, and drag and drop elements from the right menu.



4. Using Features

For custom features, including Rename, Associated Diagram, Open Linked Diagram, Generate Callgraph, and Revalidate Claim, right click on the relevant elements, and select the desired option. Generate Callgraph may be performed with nothing selected, as it affects all elements. The other features require a selected element.



- To add entry point text, double click on an element.
- To edit an element's body text, click once on the body text box.
- To resize an element, click and drag the points on the bounding box of an element.
- To delete, or refresh an element, select from the menu that appears when hovering over an element, or use the right click menu.

After using the generate call graph feature, any invalid diagram elements will be highlighted red as shown.