Midway Submission

Department of Computer Science
University of Copenhagen

Christoffer Ringgaard <cwl852@alumni.ku.dk>
Daniel Davidsen <spj290@alumni.ku.dk>
Martin Marchioro <cjr485@alumni.ku.dk>
Martin Domaradzki <wvk691@alumni.ku.dk>
Oliver Madsen <hsv488@alumni.ku.dk>
Amos Weckström <xhc484@alumni.ku.dk>
Rohan Ramachandran <sjz686@alumni.ku.dk>

July 5, 2024

Contents

1 Introduction

Dynamic Condition Response (DCR) graphs are a form of process modeling used in computer science and artificial intelligence for modeling and analyzing complex systems, especially those involving dynamic interactions and behavior. These graphs consist of nodes representing events or conditions, and edges indicating the allowed sequences or responses to those events or conditions. DCR graphs have applications in various domains, including workflow management, decision support systems, and event-driven simulations, where understanding and modeling dynamic behavior is essential.

This paper documents the creation of a piece of software that can be used to create and model DCR graphs in collaboration with other users using a fully distributed peer to peer system. As a project built for the Software Engineering and Architecture course at the University of Copenhagen, we received a "client" in the form of a teaching assistant who directed us on what he wanted his software to do. As the engineers, we designed the architecture of the software in close collaboration with our client and delivered a minimum viable product at the end of the semester.

This paper consists of four sections. The first section is a project plan, which contains basic information about how our team with communicate and specifies our definitions of developer roles according to an Agile framework and our sprint schedule for the development phase of the project.

The second section covers the requirements for the project given to us by the client. It covers functional and nonfunctional requirements as well as quality attributes for us to determine if we have satisfied the user's needs.

The third section illustrates the software architecture. We determined the level of viewability that each stakeholder of the software should have, and then created diagrams for the decomposition, uses, components and connectors, and security views. We also included descriptions of each element of the software and rationale for each decision we made regarding the architecture.

The fourth section covers our meeting with the client to determine a minimum viable product to be delivered at the end of the development phase of our project, as well as an analysis model with object and dynamic models.

The fifth and final section was written after the development phase, which lasted about 8 weeks and concluded with the production of our minimum viable product. It includes a description of our process for the development phase when taken out of theory and put into practice, as well as descriptions of design patterns and testing methodology of the final product. An appendix and references have been included as well.

2 Deliverable

https://github.com/spj290/DCR-verification

3 Project Plan

This is an overview of the project plan for the DCR project. The project will be developed using a scrum methodology, consisting of multiple sprints and close communication between team members during the various phases of development. Here we define the roles for each team member as well as set guidelines for our communication processes. This is a living document that will be updated throughout the projects as pertinent information becomes available.

3.1 Roles

For this project we will assign different roles to the team members. The roles are the typical roles that exist when working on software projects. Our team consists of 7 members, but only 5 major roles, so we decided to have 2 of each front and back end developers.

3.1.1 Role Descriptions

- Frontend Developer
  - Responsible for the UI/UX of the product.
- Backend Developer
  - Responsible for the functionality and implementation of the functional requirements.
- Product Owner/Manager
  - Communicates with the customer.
  - Is responsible for understanding the requirements.
- Lead Tester
  - Responsible for ensuring that the implementations work correctly and according to the requirements.
  - Communicates with the developers to be able to know which parts of the projects are in the test phase.
- Scrum Master
  - Ensures that the team follows the scrum process.
  - Responsible for facilitating clear communication.

3.1.2 Assignment of Roles

Our goal is to make all group members acquainted with each role in the development process, therefore we opted for a strategy in which roles are rotated at each sprint. The main drawback with this approach is that some of the knowledge each member has obtained in a given sprint is lost, meaning that the members will have to spend some time getting familiar with the newly given role at each sprint, which negatively impacts the overall efficiency.
To minimize the time spent between each sprint getting acquainted to the new roles, one could employ meetings for transferring the knowledge obtained from previous sprints.
We decided that our sprints will begin in block 4 as the responsibility of planning the project should be split equally among all group members. Each sprint will last one week as we rotate roles between group members. Each week we will have a standup meeting on Monday at 10am as well as an as-needed weekly meeting with the client on Thursdays at 15:00. We will add additional meetings as needed. Our first few sprints will primarily concern the local system of the software, then the later sprints with focus on adding peer to peer functionality.

	Frontend Developer 1	Frontend Developer 2	Backend Developer 1	Backend Developer 2	Product Manager	Lead Tester	Scrum Master
Sprint 1	Daniel	Oliver	Martin M	Rohan R	Christoffer	Amos	Martin D
Sprint 2	Martin D	Daniel	Oliver	Martin M	Rohan R	Christoffer	Amos
Sprint 3	Amos	Martin D	Daniel	Oliver	Martin M	Rohan R	Martin M
Sprint 4	Christoffer	Amos	Martin D	Daniel	Oliver	Martin M	Rohan R
Sprint 5	Rohan R	Christoffer	Amos	Martin D	Daniel	Oliver	Christoffer
Sprint 6	Martin M	Rohan R	Christoffer	Amos	Martin D	Daniel	Oliver
Sprint 7	Oliver	Martin M	Rohan R	Christoffer	Amos	Martin D	Daniel

3.2 Communication Tools

- We have chosen Discord as our primary communication tool, as everyone in the team is familiar with it. Discord also provides many different communication channels such as text communication and voice communication. It also allows for easy organization of important information.
- We will primarily be using email to communicate with our client asynchronously.
- We have chosen to use GitHub as our version control system as it is the favored software of many organizations and it provides an easy UI to interact with as well as flexibility in branch creation.
- We have decided to use Trello to track our development process as it has an easy-to-use Kanban board template to map our scrum development process to.

3.2.1 Team Communication

Team communication will occur asynchronously as needed throughout the week over Discord. 10am Monday, the team will conduct a synchronous meeting to discuss the various tasks for the week.

3.2.2 Client Communication

We will be having weekly meetings with the client to ensure that the project progress is compliant with the client's expectations. The meetings will be held primarily on Zoom, but if the team and client decide that an in-person meeting is necessary, we will schedule one on the Thursday of that week around 3 PM. Amos is responsible for communication with the client (project manager/owner).

3.2.3 Roadmap/Project Timeline

- Project Plan: Prior to the requirements elicitation, we will conduct background research to gain insight into the topics required for this project. All team members will familiarize themselves with information needed for this project, including DCR graphs, and our tools (GitHub, Bash, etc.).
- Create List of Specifications (Requirements) - March 10, 2024
- Analysis of Requirements: Rank all the requirements and divide them into Functional and non-functional requirements. - March 10, 2024
- Develop Software Architecture - March 17, 2024
- Detailed System Design - April 7, 2024
- Final Submission - June 16, 2024

3.3 Background Information

Every member of the group will be responsible for conducting their own background research so that everybody has an understanding of the scope of the project and its intended functionality. The following is a list of links of essential information that everyone should gain familiarity with. A full professional understanding of DCR graphs is obviously not necessary for this project, but some knowledge of the aim of the product is needed.
DCR Process Methodology Walkthrough: `https://www.youtube.com/watch?v=1AL8Uh7JhyI`
DCR Relation documentation: `https://documentation.dcr.design/example/case-management-process/`
[Marquard et al.(2015)Marquard, Shahzad, and Slaats]

4 Requirements Elicitation

We determined the requirements for our project by having an hour long preliminary meeting with our client. We brought a list of questions to ask to help us understand what the client's needs were. This questionnaire with our notes on the meeting filled in can be found in the appendix. We determined that we need to build this software from scratch as we have no other software to edit or build off of to suit this customer's needs, which means we are going to use a greenfield engineering model. We formed our questions with the understanding that we would need to create a completely new product.

In the meeting, we aimed to fully understand the scope of the project. Our client conveyed to us that he wanted a piece of software that could model and simulate DCR graphs, as well as share and collaborate the models on a fully distributed peer to peer network with a full set of security options. This software is to be used exclusively by professors and other professionals with a full working understanding of DCR graphs, so there is no need for our software to explain from scratch how to use DCR graphs. Much of our discussion centered around the details of collaboration as that is the most complicated aspect of the software. In the end, we came out with a strong idea of what the client's needs were and endeavored to come to the next meeting with a detailed set of requirements to present to him to confirm that our visions of the product are aligned.

The rest of this section contains a full analysis of the requirements given to us by the client. We created a functional model with a use case diagram to describe the flow of the most important tasks the user would want to complete using our software. This also shows the includes and extends relationships of every task as well as how multiple actors would interact. This is a simple abstraction that we can show to our client. The starting point of our main use cases is the following:

- The user must have a canvas on which to create DCR graphs.
- The user must be able to simulate the modeled DCR graph both manually and with test cases running in the background.
- The user must be able to able to go "online" and "offline," which means to enable and disable syncing with other users' copies of their local DCR graphs.
- The user must be able to export and import backed up copies of the DCR graph.

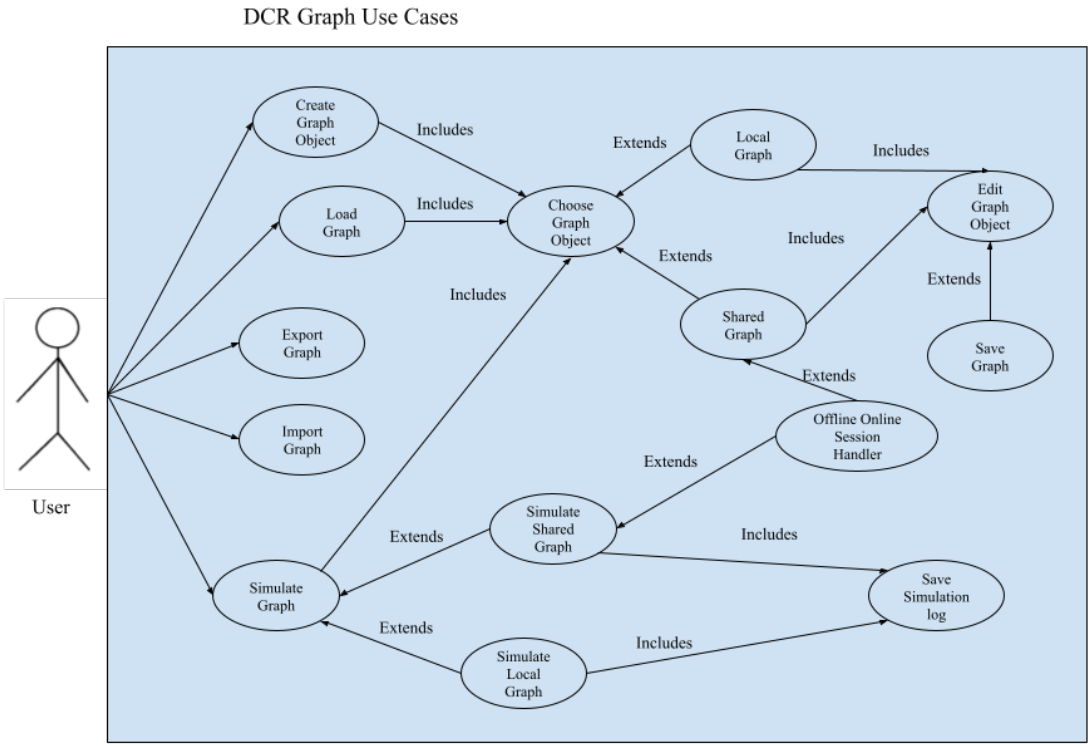
These are the main tasks a user should be able to complete with the software.

The section on non-functional requirements includes metrics and general ideas that would be beneficial to the user without being direct tasks that they need to complete. This includes ideas like responsiveness of the system and ease of use. The client has provided algorithms for us written in typescript so to avoid having to write our own DCR graph algorithms, we should create our program so that it is able to use those algorithms. The client also requires the program to be usable on Linux.

The section on quality attributes includes the main quality attributes in order of importance to our projects: usability, availability, modifiability, interoperability, performance, security, and testability. Generally, being able to use the system and keeping it available is most important to the function of the system and traits like security and testability are not as important for the use of this software. We also included a select list of quality attribute scenarios complete with descriptions of source, stimulus, artifact, environment, response, and response measure. These ensure that we have measurable goals for our software to obtain.

4.1 Functional model

4.1.1 Use Case Diagram



4.1.2 Modeling a DCR Graph

Use Case Name	**Modeling A DCR Graph**
Participating Actor	User
Entry Conditions	Application is started
Flow of Events	1. User opens the graph editor and chooses to load a saved graph or to create a new graph. 2. User selects options between creating or deleting an event or graph relation. 3. User draws various graph elements including nodes and relations, while following standard guidelines (graph first, then create relations and events). 4. If the user attempts to exit without saving the graph, warn the user about their potential lost progress.
Exit Conditions	Main menu, Graph Editor and Simulation
Exceptions	• Error in creating the Graph due to incorrect graph topology • System faces an error while attempting to automatically save the graph locally. Prompt is made to the user to ask for a manual save.
Quality Requirements	• Users are able to create a new graph with relations and events within the graph following standard guidlines.

Save, Import, Export

Use Case Name	**Save, Import, Export**
Participating Actor	User
Entry Conditions	Application has started
Flow of Events	1. User selects the Share button within the graph editor 2. User selects between manually saving the current graph, and exporting or importing the graph. 3. If the Import or Export graph options are chosen, the graph data is converted into a serialized format (e.g. JSON, XML) and saved locally.
Exit Conditions	Graph is successfuly Saved, Imported, or Exported
Exceptions	• Graph is unable to save data to users computer. • System faces an error when attempting to convert graph data to a serialized format for Import or Export.
Quality Requirements	• Graph should recognize when a user is done making changes and regularly save the graph data.

Simulating and Testing a DCR Graph

Use Case Name	**Simulating and Testing DCR Graph**
Participating Actor	User or multiple Users
Entry Conditions	User selects Simulate option on the graph page.
Flow of Events	1. User enters simulation screen for the graph. 2. If User is attempting to simulate a collaborative graph, they are sent to a offline/online session handler. 3. If user is simulating a private graph, application provides options for simulation including configuration (graph topology, node properties, relationships an d behaviours). 4. Users may choose from preset simulation scenarios to speed up the process. 5. Users are able to monitor the live graph simulation with relevent metrics presented. Users can select from options including stop, play, and pause. 6. Once simulation and testing is complete, the graph simulation log is saved to the user's local copy.
Exit Conditions	Simulation successfully completes
Exceptions	• Simulation fails and User is invited to change the configuration to attempt again.
Quality Requirements	• Users are provided with alerts in real time when simulating the graph.

Offline/Online Session Handler

Use Case Name	**Offline/Online Session handler**
Participating Actor	Users
Entry Conditions	Users select either a Shared graph or a shared simulation
Flow of Events	1. Both the user and the receiver are joined in a shared session 2. User and the receiver can both make changes in real time to either the Graph or the Simulation 3. In a collaborative graph simulation, users will be given the option to join the other user's simulation, or to continue working on their current graph. 4. In the case that they choose to join the collaborative simulation session, they will be notified when the sync has begun. 5. Both users can execute as many times as they want, and the simulation log will be saved when it is closed.
Exit Conditions	Shared graph is completed or simulation ends
Exceptions	• Error during the session hosting, sync is lost. • Unable to merge changes made by both users
Quality Requirements	• Export option must notify when the receiver is Online or must keep a notification for the receiver when the become Online.

4.2 Non functional requirements

- The users will consist of research professionals. They are expected to have prior experience with DCR graphs. Thus, it is neither required of the system to teach the user about DCR graphs, or be easy to learn.
- The user should not be able to create any "illegal" inputs. Any "illegal" graphs should be tested by user-defined tests.
- The system should have enable multiple users to work on the same DCR-graph at one time with a delay of less than .5 seconds.
- The system should be easily expandable. It should be easy to create new features such as new relations or sub-processes (if not already implemented). This means that documentation is a good idea, while not required. Further extension of the implementation is not expected to be addressed by the project's previous developers.
- It is specified that the program should at least work on Linux. While optional, it would be good if the system works in Windows too. The user wishes for extensions to be easily implementable with Typescript.
- Our DCR grapher will be used locally on a computer. There is no "maximum process power" specified for the implementation. This makes the individual computer responsible for how much process power is available for the tool. Although, it is specified that it should not operate slower than the response time of a reasonable program.
- The local file should be saved manually by the user, and the system should remind the user when closing/leaving a session. No backup system is required.
- The user should be notified if a background test takes more than 5 seconds so that they may run it themselves.

4.3 Quality attributes

Below are the quality attributes, ordered by decreasing priority

Usability We find usability to be the most important quality attribute, since this will be the primary factor that contributes to the user experience. Since we are making a tool that users are going to be utilizing to create graphs, an intuitive and user friendly GUI will result in a better user experience. If we were making a system that users would not directly interact with this might be less of a concern, but in our case we are making a tool the users will be working directly with, therefore usability is important.

Availability Availability is also of high priority since we expect the system to be accessible in both online and offline mode based on if a peer to peer session is live. Additionally, the peer to peer collaboration between users should be reliable and fault tolerant, ensuring that errors caused by one user don't propagate and affect the accessibility of another user.

Modifiability Modifiability is another major concern since the users that will be using the tool are likely to be researchers with a background in computer science, therefore the users might have the abilities to write extensions that they would find useful for their specific task. Since the users would be likely to want to write extensions it is of high concern that we take this into account when designing the system, making it easily extendable.

Interoperability The system is expected to function independently without the need for other systems, with the only exception being when importing and exporting the DCR-graphs into another format.

Performance Since we do not expect there to be any major computationally heavy parts of the system, performance is not a major concern. Perhaps in the simulation of the DCR graph but apart from that we do not see any major reason as to why performance should be that important.

Security After our meetings with the client, we've concluded that security isn't a top priority because the software is likely not going to be used for very critical projects. We're building the software from scratch, and we have a lot of requirements that are mainly about making the software user-friendly and functional. So, spending a lot of time and resources on security might take away from these important aspects that we think are more essential for the project's success.

Testability Since we will be provided with the algorithms for simulating the DCR graph, testability is not a huge concern. We will just have to make sure that the relations between nodes are working as intended.

4.4 Quality attribute scenarios

4.4.1 Usability

This quality attribute concerns the ease of the requirement of being able to create DCR graphs. Since we will be making a tool that users will be interacting directly with, usability is a major concern. This means making an intuitive UI and making sure that it is easy to be several people working on the same project. Both of these things contribute massively to the user experience.

Portion of Scenario	**Possible Values**
Source	User
Stimulus	The user attempts to create a graph
Artifact	GUI
Environment	Normal operation
Response	• It is easy to find the feature needed for each step and easy to use it to create the desired relations and nodes • The system warns the user about potential faults in the graph
Response Measure	• It takes less than 10 minutes to create a relatively simple graph • Each event and relation takes at most 3 clicks to add

This quality attribute concerns online and offline mode. This scenario is of utmost importance since it will be the primary task that the user will perform using the system, and it test how intuitive our system will be. If a user can create a simple graph in 10 minutes then the system will actually be useful for users, whereas if it takes an hour to do so, the system will not be helpful at all.

Portion of Scenario	**Possible Values**
Source	All users in an online session
Stimulus	User makes changes to online graph
Artifact	Synchronization
Environment	Online mode
Response	• User can see changes made by other users
Response Measure	• All online changes to the graph should be available to every user within one ping

This scenario outlines when a bunch of users are working in a collaborative online session. If several users make changes to the same object, then what changes should actually be kept and synchronized among the users. We propose that the newest change should be the one that should be kept. Therefore users will be able to overwrite each others work, but the changes should appear in real time.

Portion of Scenario	Possible Values
Source	User
Stimulus	User trying to export/import/
Artifact	runtime
Environment	Online/Offline mode
Response	the system performs the desired operation and does it on the latest version online or the current offline version The system reports to the user if the export was successful or unsuccessful The system prompts the user to specify the file path for importing or exporting the graph
Response Measure	User being able to export a graph in a file format that is supported by DCRgraphs.net and Draw.io User being able to import a graph in a fileformat that is supported by DCRgraphs.net/Draw.io

This scenario specifies how a user would export/import a file that is compatible with third party software. It's important for the system to be compatible with other software so that a user can easily work with the graphs in different environments.

4.4.2 Availability

Portion of Scenario	Possible Values
Source	Host crashes but other users want to continue working offline
Stimulus	Connection to the host is lost
Artifact	Document storage and synchronization
Environment	Online/Offline mode
Response	The latest received version of the graph from the host is stored This new graph can be used to host a new session
Response Measure	Receiving a waning about loss of connection, but being able to continue working on the graph if connection to the host is lost for more than 30 seconds. If a user leaves an online session they will be asked if they want to save the current graph locally before exiting

This scenario outlines how the program should handle loss of connection to the host. This is a scenario that will eventually happen and instead of it causing a total crash and loss of data, we would instead like for the user to be able to continue working on the graph. This is also an option that users should have available when they leave a session themselves.

4.4.3 Modifiability

Our goal is to develop a well documented codebase with strong modularity such that it is easy to extend without breaking the existing functionality. For our scenario we consider a somewhat simple extension, in which the user wants to extend the rule-set with a new rule type (arrow).

Different tactics can be employed to increase modularity and encapsulation, thereby ensuring the best conditions for modifiability. These include reducing the size of the modules, increasing cohesion by

assigning each module with a specific responsibility and decreasing dependencies across modules.

Portion of Scenario	**Possible Values**
Source	User attempting to add new rule type to source code
Stimulus	User requests his changes to be made public (merged into the main code-base)
Artifact	Code
Environment	Development mode
Response	If the new rule implemented by user is deemed reasonable (through thorough testing), it are accepted, merged into the main code-base and made public. Otherwise modifications are rejected
Response Measure	• The added rule successfully extends the rule set without breaking existing functionality • The added rule doesn't modify existing code, and is seamlessly integrated by acting as an independent component (modularity) • The modifications are accepted or rejected within 7 days

This scenario details when users who have programming knowledge might want to extend the system. The system should be easily extendable, which means well documented and have good interfaces with can be easily extended into new classes.

4.4.4 Security

While security is not a primary concern, basic security considerations would still be wise to implement. Definitionally there will always be some marginal cost to other features in implementing these, but we have come up with some cases where the utility outweighs the costs.

Portion of Scenario	**Possible Values**
Source	An invited user tries to invite a new user
Stimulus	User requests the owner of the graph to give them the privilige of inviting new people
Artifact	User-permissions
Environment	Online mode
Response	If the owner allows the user to gain privilige they're allowed to, otherwise their application for privilige is rejected
Response Measure	• The user is allowed to invite new users when given permission • The user is not allowed to invite new people otherwise

5 Architecture Document

5.1 Stakeholder / View table

	Decomposition	Uses	Component	Security
Users	n	n	o	o
Acquirers	o	o	m	o
Developers	h	h	h	h
Maintainers	h	h	h	h
Testers	m	m	m	m

Table 1: none, **o**verview only, **m**oderate detail, **h**igh detail

- **Users** don't need to have a detailed overview of the whole system for them to be able to use it. However they might need an overview of security to know how safe the system is
- **Acquirers** need a general overview of everything since they are the ones who commissioned the system and they would want to know what they are getting. They do not need in depth knowledge of the entire architecture of the system, but a moderately detailed view of the different components of the system will be useful to establish if the system will have the desired functionality.
- **Developers** need to be able to access and view the entire system, as they will be the ones who create it.
- **Maintainers** need a high detail view of the entire system since they have to be able to maintain every part of the system.
- **Testers** need a moderately detailed view of the system architecture since they will need to know what components there are and how they are related in order to write meaningful tests.

5.2 Combined views

5.2.1 Decomposition & Uses View

We have chosen to combine the decomposition and uses views due to numerous shared elements and stakeholder interests. The rationale behind this decision is due to them both comprising of modules. The decomposition view is made up of the overarching hierarchy of module parent-child relationships.

The uses view is comprised of relationships between the various modules. We believe that these views are best described together, as they are mutually exclusive and require one another to have a high level understanding of the application architecture. Therefore, we choose to combine these views to provide stakeholders with a necessary understanding of the hierarchy of modules as well as the relationships between them.

5.3 View Documentation

5.3.1 Decomposition & Uses View

Primary Presentation Diagram

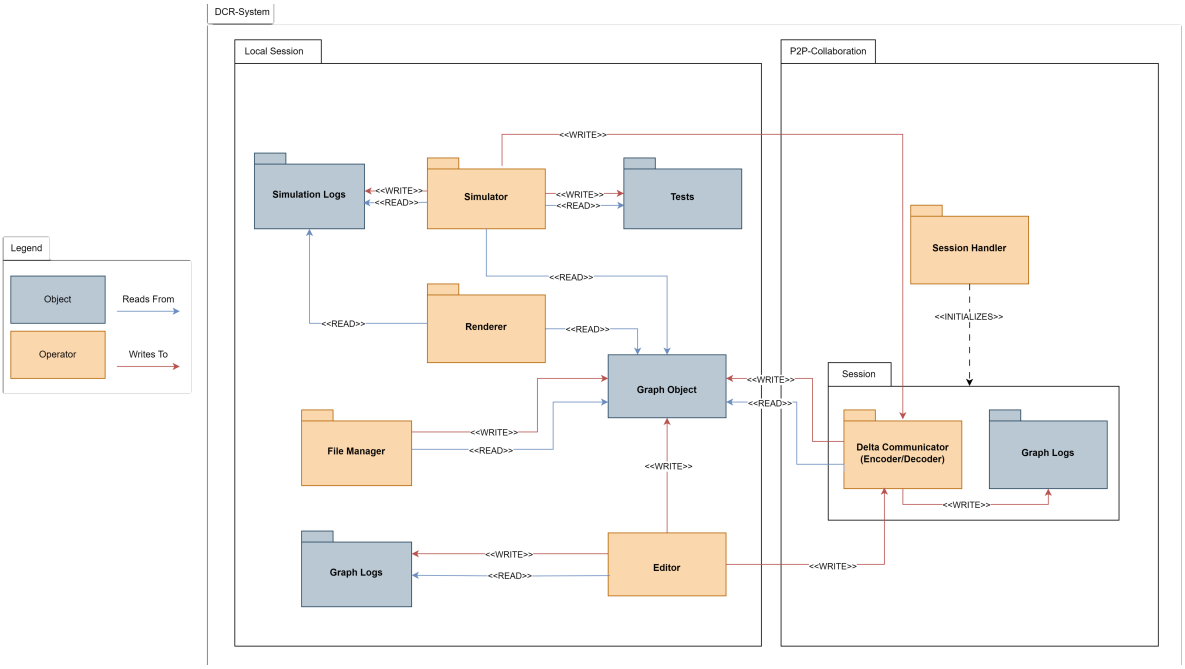


Figure 1: The primary presentation diagram of the combined Decomposition/Uses view

Element Catalog

The DCR Grapher is split into two sections. There is the software that only runs locally, as well as an extension that runs when peer to peer editing is enabled. The pieces of the local software are as follows:

- Object - Submodules that are blue are Objects. These are parts of the software which represents some sort of data, and does not contain any functionality.
- Operator - Submodules that are orange are what we call Operators. Operators are submodules that contain the logic and functionality of the system.
- Graph Object - Everything stems from the central graph object, which is a file stored on the local computer representing a given DCR graph. There may be many of these.
- Editor - This item operates on the graph object, making changes to it from the user input. It also writes and reads the graph history for version control purposes. It also writes changes to the delta communicator when online mode is enabled.
- Simulator - This reads the locally stored graph and handles the user running simulations on it. It also reads the automatic user defined tests and runs those in the background. It also writes and reads the user's currently run simulation to the simulation log in case the user wants to save their simulation to run it again quickly. It also writes new tests to the delta communicator when online mode is enabled.
- Tests - This contains user defined tests.
- Renderer - This reads the locally stored graph and creates it into a user friendly image on the screen. It also reads the operations of the simulator to display the current step of the simulation.
- File Manager - This handles saving and loading of the graph as well as importing and exporting the graph into file types for the purpose of creating backups or viewing the graph in PDF form.

- Session Handler - This controls the peer to peer session when online mode is enabled.
- Session - A session contains two parts. First, a delta communicator, which encodes and decodes changes to the graph across the network rather than sending the entire file between peers upon every change. It also contains a global graph log that tracks all of the changes to the graph by all users, not just the local user.

Context Diagram

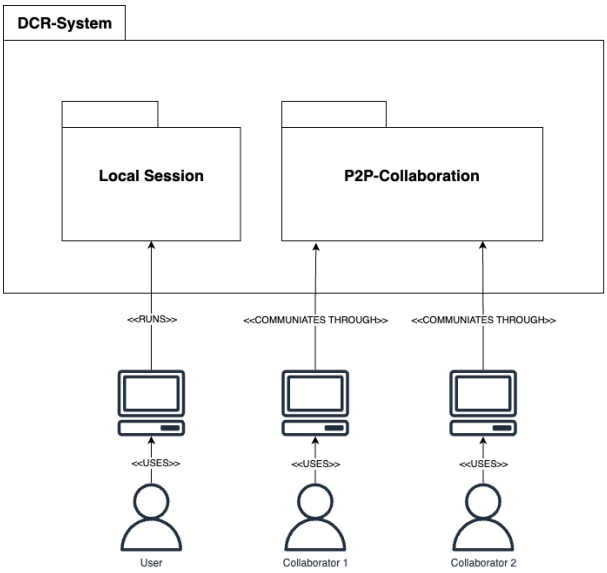


Figure 2: The context diagram of the combined Decomposition/Uses view from the point of view of a user who is running the program with a bunch of collaborators

Variability Guide

- Optionality of online status - The session handler and its session only exists when the user enables online mode. Otherwise, the only part of the program that exists is the local session.

Rationale

Our reasoning for forming our software this way was centrally designed around the ability for the program to support peer to peer collaboration. Because our client requested that a user be able to switch between offline and online mode, we included functionality that works at all times which we called the local session, and functionality that is enabled only when online mode is enabled which we added in a P2P-Collaboration section. The idea of organizing the software around the central graph object reflects the goal of the software – to create DCR graphs. As a result, our editor, renderer, and simulator all work mainly on the graph object and include some logging on the side to improve the user experience. When enabling online mode, a session handler is created. For the host, this opens up access for other users to access their graphs. Other users acquire a copy of the graph, which is operated on by an editor, renderer, and simulator all the same. Changes to the graph are communicated through the delta communicator, which encodes and decodes changes to the graph sent across the network. The key part of the peer to peer exchange is that it communicates only changes to the graph rather than sending the entire graph after every change, which makes the program responsive regardless of the size of the underlying graph. As a whole, the tool is effective and responsive both online and offline.

5.3.2 Component-and-Connector

Primary Presentation Diagram

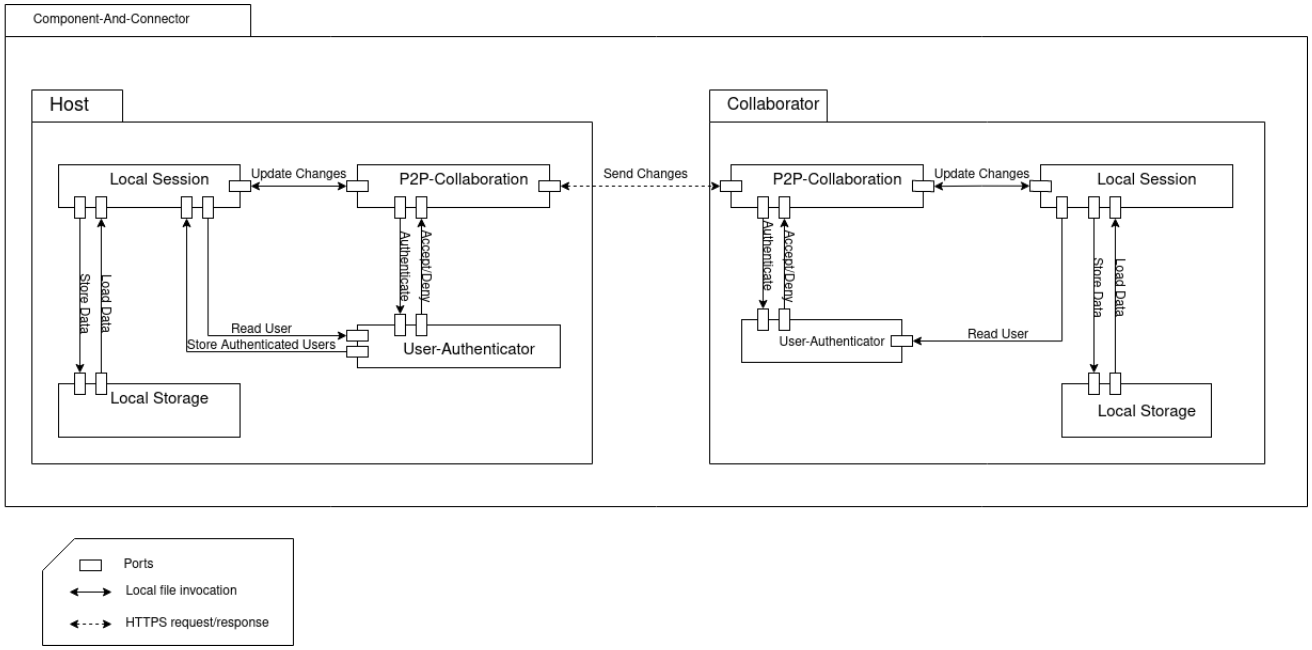


Figure 3: Components-Connectors view of the system

Element Catalog

Our component-and-connector view is split into the two type of users of a collaborative session, the host and the collaborator. This is due to the peer-to-peer functionality of our application. The host and collaborators make use of almost identical components, only exception is that the collaborator is not able add users to the session. The shared components between that user and the host has the same functionalities.

- Local Session - This is where most of the processing is handled. The user can in here model a DCR-graph and run simulations/tests.
- Local Storage - Local file component, where all essential data for the system is stored.
- P2P-Collaboration - This component is designed to facilitate real-time collaboration between the host and any authenticated collaborating users. We have chosen to design this component in a way that it will keep track of any new changes to the shared file. Additionally, this component will send an authentication request on the host user's side to ensure that the collaborating user is authenticated before any changes are made to the graph or simulation.
- User-Authenticator - A simple checker that checks whether a received message from User is a valid user with write-access.

Context Diagram

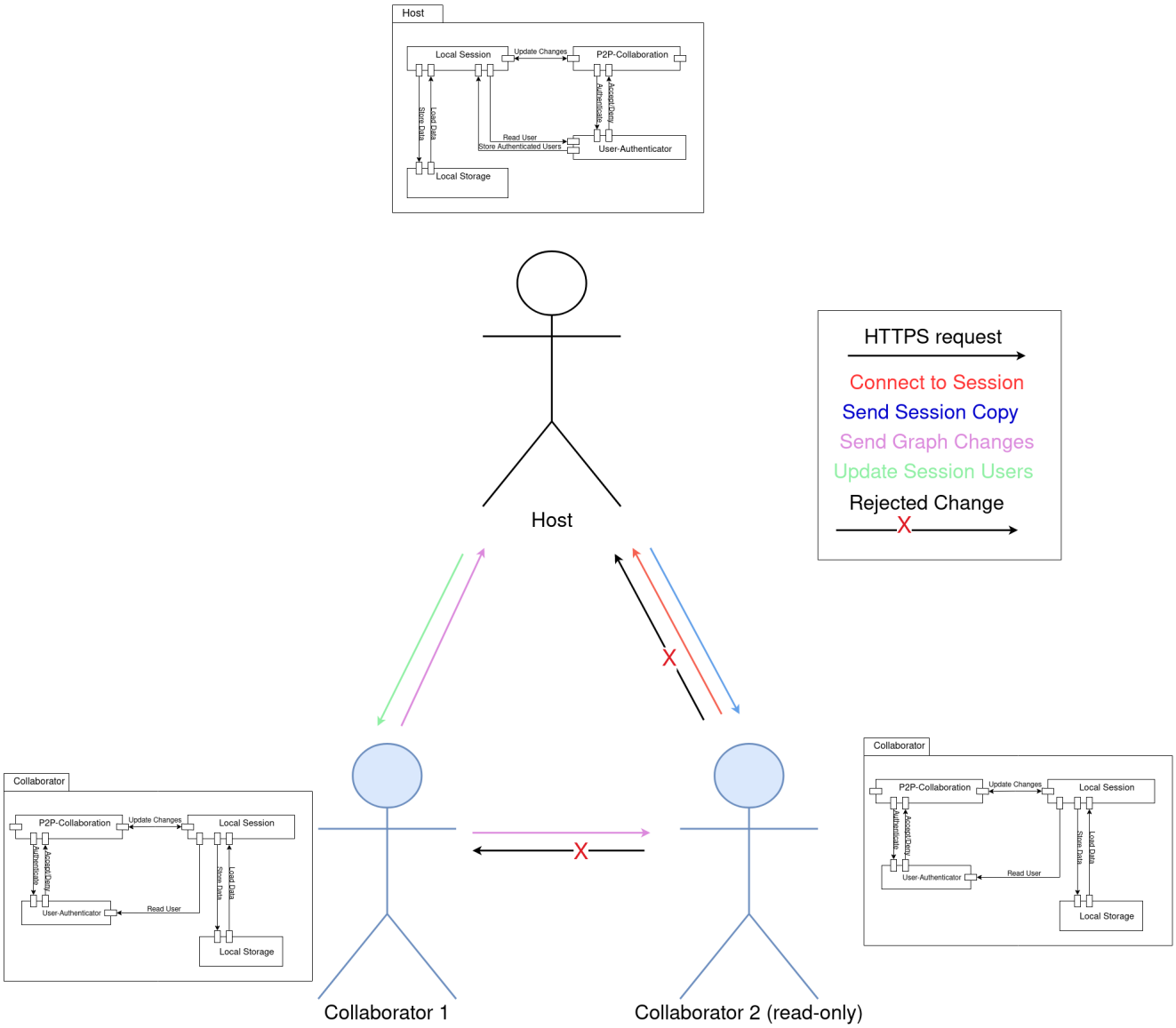


Figure 4: Collaborator 2 connecting to a session with read-only access. Then both collaborators tries to make an update to the graph

Variability Guide

The local components (storage and session) are in use for all users. These 2 components are the bare bones of the program. Only when a user decides to host a session, the P2P-Collaboration and User-Authenticator are set in use, while the User-Authenticator is only used by the host. The graphs shows how the collaborators only communicate with the host, while the host is responsible for broadcasting any accepted changes to the graph.
A user might also choose to only use the Local Session. If a user decides not to save anything, the local storage is never used. This is also a possibility for the collaborators. They might have joined the session, but never save a copy of the DCR-graph. This way the collaborator's local storage is never in use.

Another variation point is the collaborators permission level. The host might give some users read only access, while others will additionally have the ability to write. These different roles are depicted in the Context Diagram (Figure 4).

Rationale

The represented components showcase the main part of the system. The Local Storage and Local Session keep track of any changes made to the session data, as well as save and load requests. In the case of the host user running a DCR graph solo session, the Peer-to-Peer (P2P) collaboration component and the authentication component will not be used to collaborate and authenticate users. Instead, once a collaborator has joined the session, any changes will go through these elements for authentication. We have designed the components in this manner to simplify the task of providing both solo and collaboration functionality to the application.

In the case of a user hosting a session with multiple users, the P2P collaboration component is added to ensure functionality for multiple users. We have designed this component to only keep track of changes from either the collaborating users or the host. Additionally, the host P2P collaboration component will be responsible for resolving any conflicts between the host and collaborator. The P2P collaboration component also requires the help of the authentication component to run properly. To ensure the security of the system, only the host user can choose to accept or deny any collaborators to the system. Once a user has been authenticated/accepted, then their changes to the file can be loaded/saved to the host user's data.
Essentially, the host has the functionality of a server. The P2P connection is only between host and collaborators. The collaborators are never communication directly with each other.

5.3.3 Security View

Primary Presentation Diagram

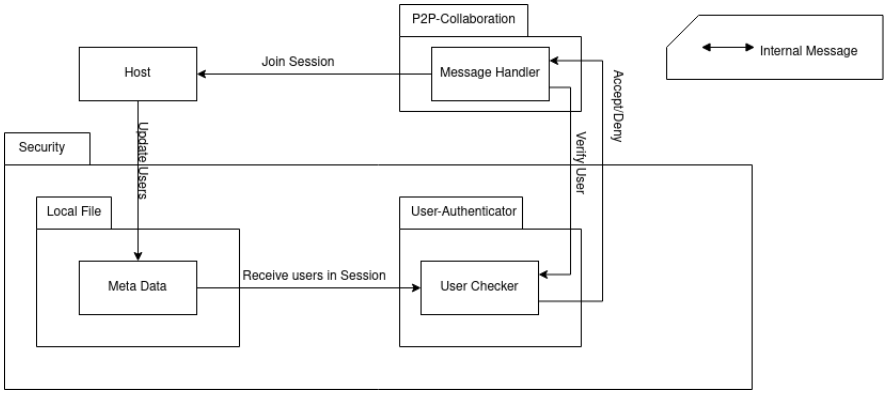


Figure 5: Security view of the system

Element Catalog

There is next to none security in our system. The security consists of confirmation of users before a change is made. Only the host is able to add users to the Meta Data.

- Host - This is the host session. Note that the security is also contained in the host
- Message Handler - Here a message is decoded and split into a message and a user. The message is either a graph-/test change or request to join a session.
- User Checker - This is where a user is identified. A change request is accepted if the user has write-access.
- Meta Data - The meta data contains information about the graph. The part used in the security is authenticated users. The authenticated users are split into users with read access, and users with read/write access.

6 Analysis

6.1 Features priorities

In collaboration with the client, we identified the following ranking of features. Features were determined based on the decomposition view. The line represents the minimum viable product features for the application.

1. Graph Object
2. Editor
3. Renderer
4. Simulator
5. Tests
6. File Manager
   (a) Save the graph
   (b) Load graph from file

---

7. Peer to peer collaboration
   (a) Change to message encoder
   (b) Message to change decoder
   (c) Communicator
   (d) Session handler
   (e) Access control (write/read-only)
   (f) Authentication Protocol
8. Simulation Logs
9. Local Graph Logs
10. Session Graph Logs
11. File Manager
    (a) Convert file format for import/export

6.2 Analysis Document

6.2.1 Short Description

We considered many factors when creating our analysis model. Primarily, we focused on building out the features that were both essential to our program, and doable within the time frame of this class. As a result, we primarily selected features essential to the application like graph modeling, rendering, simulation and testing. Additionally, we chose to include features within our file manager such as saving a graph, as well as loading a saved graph into the editor. These features comprise what we believe to be the minimum-viable-product (MVP) of our application.

We further categorized features under nice-to-have (NTH). Most notably, a peer to peer collaboration component that will allow the application to securely communicate changes in shared graphs between users. Additional functionality provided by this component includes access control, allowing a host user to set permissions for other users of a shared graph (write/read only). We also made a decision to add log functionality for both simulations and graph changes under NTH, due to them not being completely necessary for application functionality. Finally, we have defined a feature that allows import/export of graph data in a JSON/XML format, to allow ease of sharing graph data between users

outside of the application.

For our object and dynamic models, we designed the diagrams to encompass our plans for primarily the MVP features. Our object model diagram centered around providing graph editing and simulation features. In the graph editing section, we chose to use user input to allow the user to create a canvas of objects. When objects are created within the graph object, we will use a renderer class to display user created objects within the GUI. We additionally used an event class to handle user requests to change the graph object. Finally, we provided graph loading and saving functionality to allow users to save graph progress and return to graph editing when they choose. With these classes, we plan on providing full graph editing functionality through user input.

The simulation features provided by our application will be handled by the simulation, test, and tests classes. One portion involves the simulation of the graph itself, including checks as to whether the graph itself is valid. The simulator class communicates with the test class to ensure the application keeps track of simulation test results and then provides them in a list to the user through the tests class.

Our dynamic models center around describing our simulation/test class and the Peer-to-Peer collaborator communications. The simulation/test class interactions are designed to keep track of user simulation results and return them in a meaningful way to the user. The Peer-to-Peer collaborator is listed under our NTH features, but we included it regardless as it is an important part of our application vision. Communication in our Peer-to-peer component is focused on keeping track of changes made by different users and broadcasting said changes to all users to ensure everyone is working on an up-to-date version of the graph. In our implementation, we will also include user permissions to allow hosts to define access for other collaborators.

6.2.2 Object Model

We chose to model the classes of our minimum viable product in a UML diagram. We might update this later on if we have time to implement additional features.

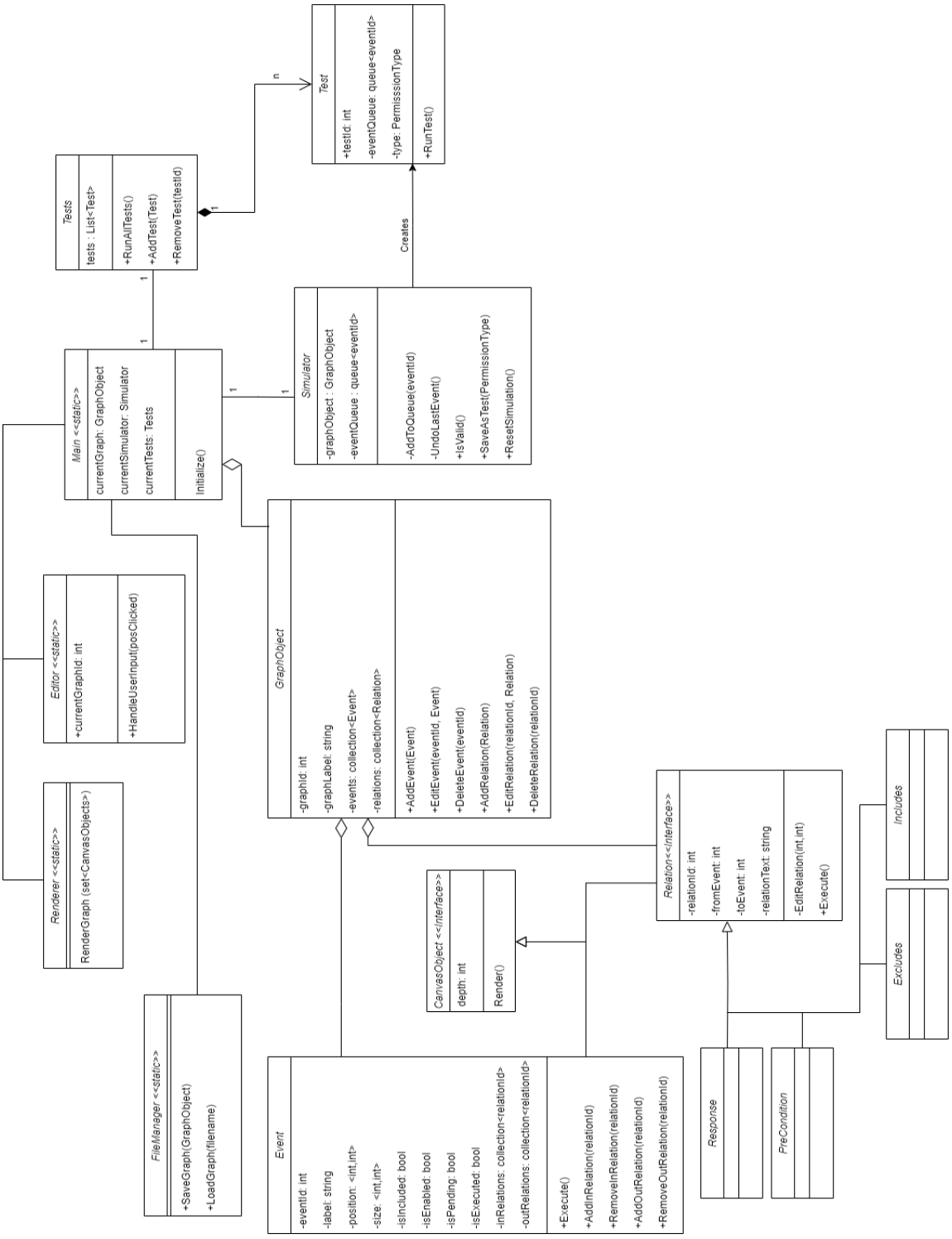


Figure 6: UML class diagram of the minimum viable product components

6.2.3 Dynamic model

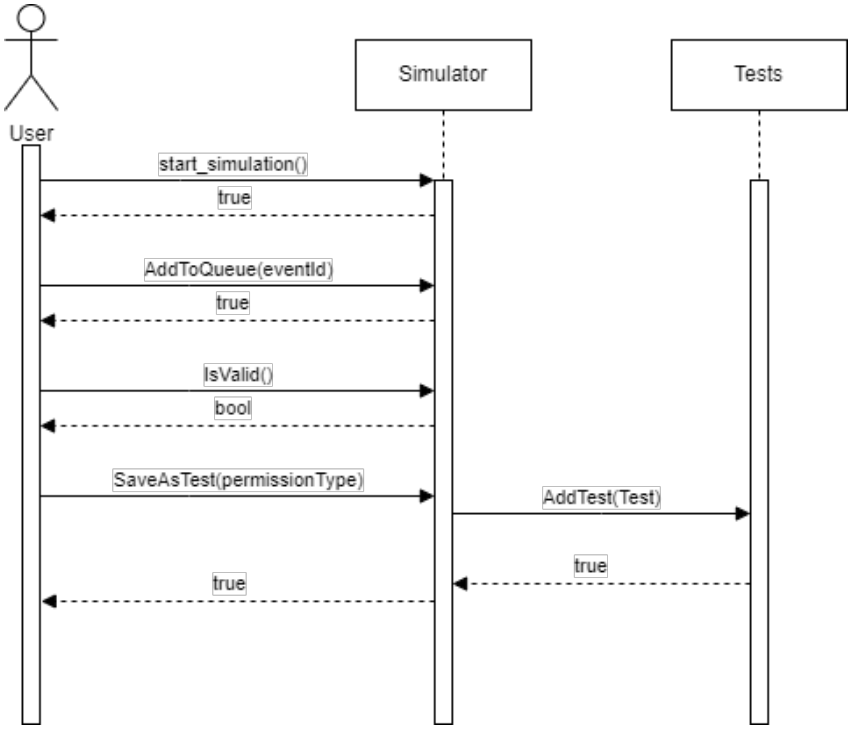


Figure 7: Sequence diagram for creating a simple test in the simulator

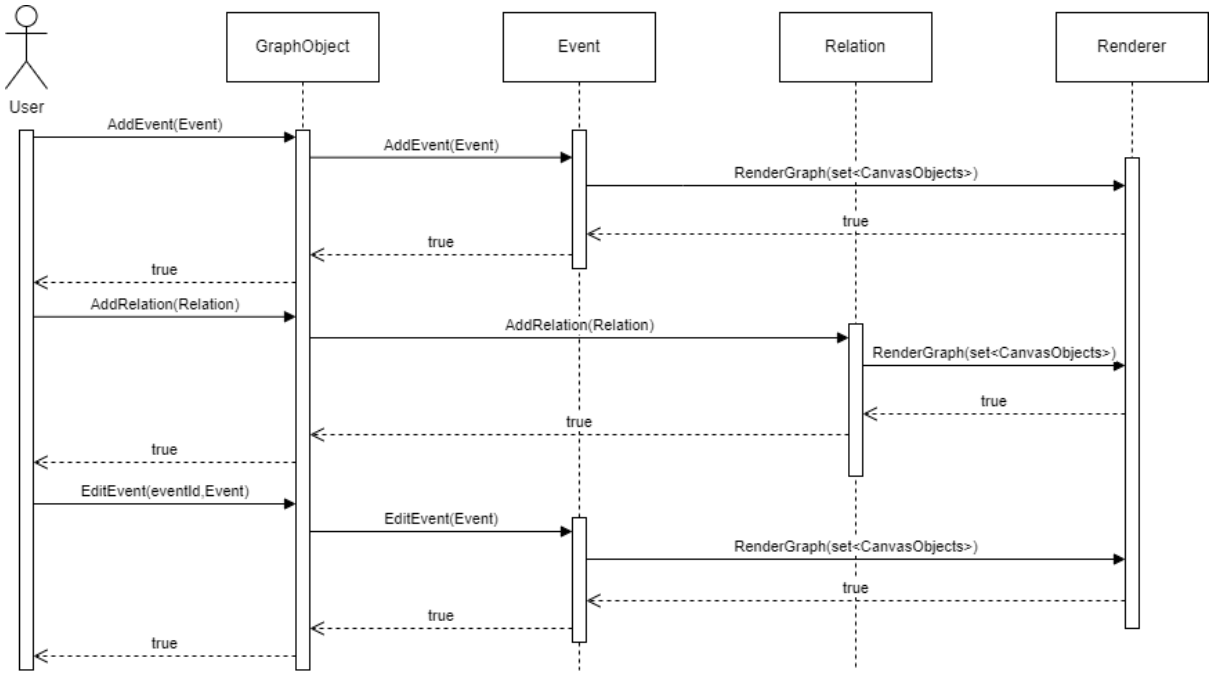


Figure 8: Sequence diagram for adding and editing events/relations

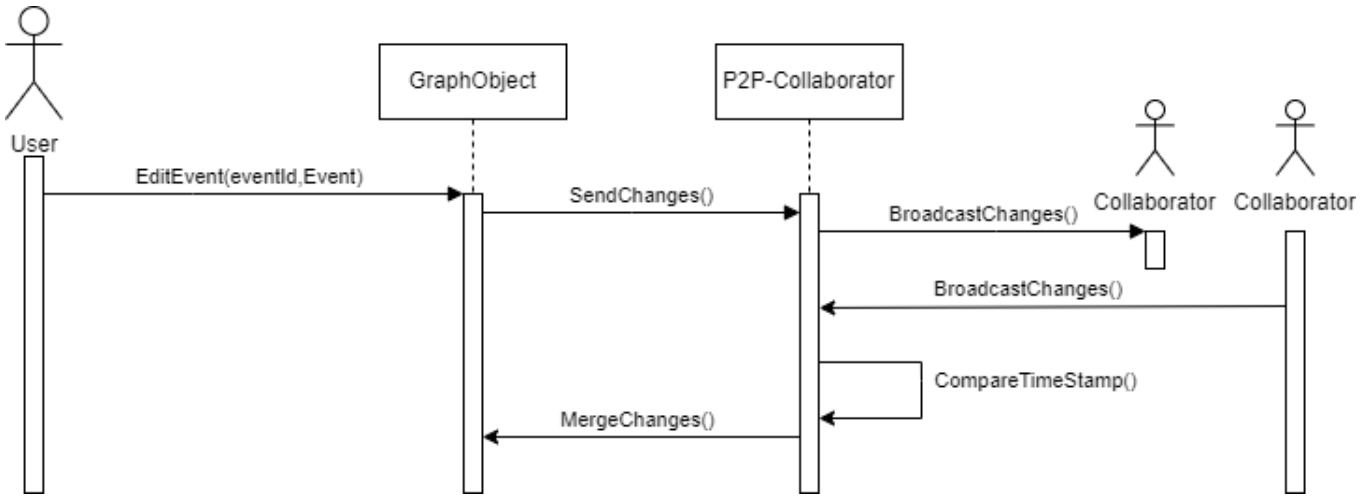


Figure 9: Sequence diagram for synchronous changes in shared graph model

7 Final Report

7.1 Development Phase

This document completes our documentation on the development on the DCR tool. The product was developed over 16 weeks, split equally between the two blocks into planning and development phases. During this time, we succeeded in implementing the minimum viable product specified in our analysis section.

We addressed the development phase as a series of steps that brought us closer and closer to our final product. We began with discussions on which languages would be best to use, and decided on a React web app with a back end using C# connected through an API. We planned on the user interface being handled by React and any calculations being done with C# as we assumed it would be more suited for the job. We all started with different approaches to try to create the skeleton for our product. However, the most successful prototype ended up being entirely created with React, and our major development operation began off of the back of this prototype. It was also during this start-up phase that we realised that the Client already had provided us with the back-end to write the code. This made multiple sections of our planning phase redundant. To avoid this in future projects, we would have to spend more time studying tools and material provided by the client.

The rest of the block consisted of a series of weekly or biweekly sprints, depending on everybody's availability. Our team met and discussed the state of the application, and then decided on the course of action for the day to bring our application to our goal of the minimum viable product. Thus, instead of working in a traditional Agile model where the team members work on their own and meet for daily standups over the course of a 1 or 2 week sprint, we condensed our sprints into a full day of working together on the project, as this was what worked best for our development process. Between these biweekly sprints we made sure to keep in contact with our client, and ask questions about the product. We would use this as a guideline for any future work we would do.

Our focus during the development phase was on creating the editing functionality of the DCR tool. The user experience was of utmost importance to us, which follows from our decision to choose usability as the most importance quality attribute of this project. Our first tasks were to create a smooth canvas where events and relations between the events could be easily created and moved. We then moved on to creating a page for choosing events and running simulations as a way of manually running tests. We then created a way to save tests and have them run in the background, clearly showing whether or not they passed with a red or green background. A lot of focus was put on improving UI and animation of the page to improve usability for the user. We included many small features such as saving and loading a graph to and from the disk in a .json file, clearing the canvas, a local version control system, and saving the current graph to cookies to ensure that progress cannot be lost if the page is closed. We put a lot of focus into ensuring that the testing suite contains all the options needed to make it a true DCR graph tester, including our client's code for testing and allowing each event to have all of the necessary attributes such as pending, included, and excluded.

Our requirements have been achieved with a great deal of success. The main use cases relating to the local graphing utility detailed at the beginning of our requirements elicitation have been fulfilled and the program achieves what a professional user would need to create and simulate DCR graphs. During our last meeting with the client, the client expressed satisfaction, that he got a product beyond expectation. Our ranking of quality attributes guided our development throughout the entire process. The usability is very high. It is possible to create a simple graph, with two events and a relation between them, with only 5 clicks. The user interface is simple but clear, with all of the options for use clearly available to the user. The program is available on Windows, iOS, and Linux, as long as the user's computer is functional due to be an entirely local system. It modifiable in that it is relatively easy for someone who can read code to add new types of relations and otherwise search through the code for something they may need to add.

Our client expressed that the actual implementation of the peer to peer functionality was strictly optional, because it would be needlessly difficult for our small, inexperienced team to implement a real time multi-user version control system on the level of google docs in such a short time. We were excited

to have a chance to work with such a system, but in the end, we did not have time to implement the peer to peer functionality detailed in the architecture document. However, we are satisfied with the result of our first attempt at implementing real software development practices, and believe we have created a very functional program for use by research professionals interested in using DCR graphs.

Despite our success, there are many aspects of the development process that could have been improved if we were to do this project again. Our detailed design phase, in which we discussed specific implementation details such as framework and programming language, was very haphazard, and the first prototype was more of a result of lucky individual tinkering than solid planning. There are numerous bugs remaining in the code that may have been squashed at creation if our testing with each new feature was more thorough. Our progress was quite slow due to meetings being declared at random in practice rather than to our set schedule, but that may be chalked up to the busy nature of the lives of seven students. Had we been able to improve our process, our product may have had less bugs and more features, maybe even including the peer to peer framework detailed in our architecture phase. However, our current product is functional in all the ways it needs to be and we are satisfied with the result, even if we could not go as far beyond the scope of the requirements as we wished.

7.2 Design Patterns

During our planning phase, our initial goal was to develop the back-end code using C#. We planned to create abstract classes to represent various features of our application. Primarily, we were concerned with using the design patterns of inheritance and object-oriented programming to build the framework for our application. The idea was that our front-end UI would be able to communicate with back-end objects to provide users with DCR graph functionality. However, as the project evolved, we decided to pivot to a completely React-based implementation. This change better aligned with our project requirements. In the React environment, we employed several design patterns to enhance the functionality and maintainability of our application. Since our application was almost entirely implemented in React, we leveraged design patterns intrinsic to React, including Components, State Handlers, and Hooks.

Component Composition was fundamental to our approach. This design pattern allowed us to construct complex user interfaces by composing smaller, more manageable components. For example, we integrated components such as Events, Relations, and ContextMenu into a larger Canvas component. This modular strategy ensured that each piece of the user interface was both manageable and reusable. We utilized the Canvas component as a container component, responsible for managing the application's state and passing it down to child components as props. This approach ensured a clear separation of concerns. By separating data fetching and state management from presentation logic, child components could focus solely on rendering user interface elements based on the props they received. The Canvas component handled the overall layout and state of the application, while child components like Events were dedicated to specific tasks, such as displaying events or managing relations between items. This modular approach simplified child components and made the application easier to debug and test.

We also employed the Callback Pattern with functions like setContextMenu, setHistory, setRedoStack, and setWindowWidth to manage state updates. In this pattern, functions are passed as props to child components. These child components invoke the functions to trigger state changes in the parent component. This bidirectional communication allows child components to notify their parent components about user interactions or other events, resulting in dynamic and responsive user interface updates.

In addition, we utilized React's built-in hooks, such as useState, useEffect, and useCallback, to manage state and side effects. Custom hooks allowed us to extract and reuse component logic across multiple components, promoting code reusability and simplifying maintenance of the codebase. For example, the useCallback hook is used to memoize functions. This is helpful to prevent unnecessary re-renders when passing functions as props to child components.

We implemented the Command Pattern in the saveToHistory function, which encapsulated state changes within a function. This pattern involves creating objects that represent actions, allowing us to parameterize objects with different requests, queue or log requests, and support undo/redo operations. By encapsulating state changes in a function, we managed the application's history and enabled features like undo and redo. These features enhanced the user experience and ensured state consistency.

Collectively, these design patterns contributed to a robust, maintainable, and scalable application architecture. This architecture not only simplified development but also made the application easier for developers to follow and for clients to understand.

7.3 Testing

Over the course of the 8-week development period, testing and debugging played an integral role in ensuring the quality and functionality of our minimum viable product (MVP). Our approach to testing was largely influenced by our Agile methodology, emphasizing continuous integration and feedback.

Throughout the development phase, our team adopted a manual and iterative testing strategy. Each sprint cycle included dedicated testing periods where we meticulously examined new features and bug fixes. This process allowed us to identify and address issues promptly, ensuring that the development progress was steady and aligned with our project goals.

Our testing strategy encompassed the following key activities:

7.3.1 Feature Testing and Break Testing

As new features were developed, they were tested to ensure they met the specified requirements. We focused on validating the core functionalities of the DCR tool, such as creating and managing events, relations, and running simulations. Each feature was tested individually and in conjunction with other features to ensure seamless integration and operation. Additionally, every now and then, we intentionally attempted to break the software to identify potential weaknesses and ensure robustness.

7.3.2 Usability Testing

Given our emphasis on usability, we conducted extensive usability testing to ensure that the user interface was intuitive and efficient. We tested the user experience by performing common tasks, such as creating simple graphs, running simulations, and saving/loading graphs. Although we made significant strides in usability, particularly in the editing functionality, we recognize that more time could have been spent making the simulator part of our software more intuitive. Additionally, the inclusion of a user manual within the software would have further enhanced user experience.

7.3.3 Unit Testing

Unfortunately, no automated unit testing has been done for this project due to time constraints. Unit testing involves dividing the application into separate independent components (units) and testing the functionality of each in isolation. This can be done both using a black box approach to ensure correct input/output behaviour, and white box to make sure that all possible branches and paths are reached and function correctly. Our approach would then be to test each function in each React component. Since most of the functions operate by creating side effects to the application state, we would need to incorporate mocking while testing, with tools like React Testing Library and Jest. We would then be able to create a mock version of the state and functions, which we can manipulate in a predictable way. The following pseudo-code snippet is an example of the overall structure of a potential unit test for deleting an event:

```
#tests if deleteEvent correctly deletes the event
testDeleteEvent() {
    // Mock data
    const eventToDelete = {
      id: '1',
      label: 'Event 1',
      ...
    };
    const initialEvents = [eventToDelete, { id: '2', label: 'Event 2', ... }];

    // Render Canvas component with mock data
    const canvas = render(
```

```
      <Canvas
        events={initialEvents}
        ...
      />
    );

    // Call deleteEvent function
    canvas.deleteEvent(eventToDelete);

    // Assertion
    // Check if the event '1' is removed from events
    expect(canvas.events).not.toContain(eventToDelete);
}
```

7.3.4 Integration Testing

Integration is a step between unit testing and system testing, in which multiple units are tested together to determine if they work as a group. We did not have automated tests for this type of testing, but it is the most natural form of testing as developers working on certain features of the product. In the process of creating new features, we often opened the software to the location of our new feature and tested its various functions and surrounding functions. For example, when improving the appearance of self-references, a developer might also test to make sure that relations to other events still look correct before determining that the feature has been successfully added. Although we did not have an explicit methodology for integration testing, it did come up in many instances of testing while adding new features.

7.3.5 System Testing

System testing involves testing an integrated system to verify that it meets specified requirements, evaluating its compliance by testing it as a whole. Unfortunately, we did not employ automated system testing in our project due to time constraints and resource limitations. However, we did ensure that all required functionalities worked as promised by manually interacting with the system. Towards the end of the project, all group members participated in creating, simulating, and testing various graphs to verify that we obtained the desired outputs from our application. Additionally, we tested various configurations, UI changes, and save/load features to ensure a complete project.

System testing would typically involve validating the overall behavior and performance of the fully integrated software product, including its interaction with external interfaces, performance under load, and behavior in different environments and configurations. Key focus areas would include functional testing to ensure that functionalities work as intended, performance testing to evaluate the system under various conditions, security testing to assess vulnerabilities, usability testing to ensure user-friendliness, compatibility testing across different devices and environments, and regression testing to ensure new changes do not affect existing functionalities. Despite not implementing formal system testing, it remains crucial for uncovering defects not apparent during unit or integration testing, ensuring a robust and reliable final product. In future projects, incorporating system testing would be essential to deliver thoroughly vetted and high-quality software.

7.3.6 Client Acceptance Testing

In the final phase of our project, we conducted an acceptance test with our client to verify that the system met all the outlined requirements. This test involved a comprehensive review of the DCR tool's functionalities, focusing on the key use cases identified during the requirements elicitation phase. The client's feedback was instrumental in validating the product and ensuring it was ready as an MVP. The client expressed great contentment both regarding the functional and non-functional requirements.

7.3.7 Bug Tracking and Resolution

Despite our best efforts, the software is not entirely free from bugs. We employed a systematic approach to bug tracking and resolution, documenting all identified issues and addressing them based

on their severity and impact. This process involved prioritizing critical bugs and ensuring they were resolved before the final acceptance test. Non-critical bugs have been documented to provide a clear understanding of the current limitations of the software.

7.3.8 Limitations and Future Work

While our testing efforts were extensive, the scope was constrained by the MVP nature of the project. Certain advanced features and potential edge cases were not fully tested due to time constraints. These areas have been identified and documented for future reference. As this project will not continue beyond the MVP, there will be no further development or testing by our team.

In conclusion, our testing phase was marked by continuous and collaborative efforts to ensure the quality and functionality of the DCR tool. The combination of manual testing and acceptance testing provided a robust foundation for our MVP. Our agile approach facilitated prompt feedback and iterative refinement, contributing significantly to the overall success of the project.

We consent to our report being used as teaching material in the future.

We consent to (parts of) our source code being published under a free open source license.

8 Appendix

8.1 Requirements Elicitation Questionnaire

Requirements Elicitation

1. Who are the users/stakeholders of the system?
    a. Users will be research professionals. It should be people who already know how DCR-graphs work.
    b. There won't be any stakeholders. The code should be open source.
2. Is there a hierarchy of users? I.e. Are there different roles that users can have? And do the roles have different permissions of what they can do?
    a. There will be collaborative sessions of the tool. There should be some different roles. Some should have read-only permission, while some will be able to write.
    b. The owner of the project should be able to invite collaborators to edit/view.
    c. The implementation should be fully distributed (peer-to-peer).
    d. Editors should get a local copy that is updated so that everyone's copy matches. (So does read-only users. But they can not push edits).
    e. Only the owner should preferably be the only one who is able edit the graph's role permissions (but not really possible).
    f. Protecting local copies more than keeping data confidential because that's not possible in p2p.
    g. There should be a feature that enables importing/exporting. Also ctrl-z ctrl-y access. You shouldn't be able to undo other peoples' changes. Change history is changed when somebody else is editing.
    h. Load/save/import/export
3. Can you give a brief overview of what a typical user should be able to do within the system? USER STORIES
    a. Eg. User1 logs in to the website, then they can then choose whether to modify an existing graph or create a new one. To create a new graph the user has to input….
        i. Swarm database, swarm note module, the graph should contain handles with edit rights (rights are saved locally). Starting a new session should start broadcasting to saved handles.
    b. When modifying an existing graph the user can delete/change the graph.
    c. User Story:

        i. Landing Page (optional)
        ii. Make a new graph or load a saved graph (continuing a collaborative session).
        iii. Go online or offline. Disabling sync in the distributed network
        iv. Model a DCR Graph, define test cases as online or offline.
        v. Simulation of the DCR graph. Either by themselves or by inviting other users to simulate the DCR graph. Simulation is running available activities/events, to progress to a new state.
    d. Tests should consider sequences of events within a context of enabled activities. Positive (there exists a legal state) and negative (there does not exist) test cases.

4. What are the most important use cases, i.e. what functionality is absolutely necessary for the system to work.
    a. Test driven modeling “stuff”. Defining test cases, creating DCR graphs, simulating DCR graphs
    b. At all times see a list of which ones failed and which ones passed?
    c. Anything not modeled is allowed (open world principle) test cases will be fine but tell user that things aren’t modeled
    d. There should be activities and labels. Activities should be unique, but multiple labels can refer to that activity. Labels are irrelevant, but IDs and relations are unique to each node. Tests are tied to activities, relations are tied to events.
5. What inputs does the user provide the system? (When they are dcr graph modeling/simulating)
    a. Moving events around (existing relations should follow the movement)
    b. New events
    c. (optional) New sub-processes
    d. New relation between two nodes or one node and itself
    e. Save/load/import/export
    f. (Simulation) Clicking on events that you want to simulate (executing
6. What outputs does the user expect from the system?
    a. The modeled graph

    b. Whether specified test cases passes or fails

7. Should data be stored, and if it should, what data should be stored?
    a. The graphs themselves (includes metadata about the graph positioning)
    b. (Metadata)Some sort of info on the ongoing collaborators
    c. Every user has a local copy of everything that is then synchronized with the cloud?? No, if a user turns sync on then it checks with every other user's copy of the graph
    d. Graph is represented as a
    e. For exporting graphs there are preferred datatypes.
8. Does the system have to know who the user is? I.e. should the user be able to login with credentials?
9. What are the minimum requirements for creating a DCR graph? I.e what are the most necessary attributes of a graph for it to be considered a DCR graph. (rank the specifications)
    a. Just nodes and relations, no subclusters yet. Each node has event id, label, role associated with it.
10. Are there any specific UI/UX considerations or preferences for the system?
    a. Eg. It would be nice if I as a user could highlight a process text to identify Roles, Activities and Rules….
11. What would a successful end product look like/ be able to do?

Nodes consist of:

EVENT_ID
ROLES
LABEL

Swarm Database?

Every user can go online to a swarm session, pick a handle, and stream their actions to the session?

All DCR Graph execution semantics already implemented in TS.

References

[Marquard et al.(2015)Marquard, Shahzad, and Slaats] Morten Marquard, Muhammad Shahzad, and Tijs Slaats. 2015. Web-based modelling and collaborative simulation of declarative processes. In *Business Process Management*, pages 209–225, Cham. Springer International Publishing. 6