Frontend Implementation of Simulation and Reachability

A Contribution to the ECDAR Project

Project Report

Group 13

Aalborg University

Department of Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
DK-9220 Aalborg Ø
https://www.cs.aau.dk/

Title:

Frontend Implementation of Simulation and Reachability

Theme:

Complex Backend Software

Project period:

September - December 2022

Semester & study:

Software, 5th semester

Project group:

CS-22-SW-5-13

Participants:

Andreas Paludan
Casper Zacho Roskær
Emilie Sonne Steinmann
Ibrahim Alaiddin Abu Rached
Julie H. Bengtsson
Mikkel Dolmer

Supervisor:

Martijn Goorden

Abstract

Software updates are an inevitable part of modern day lives – both providing new features, but also correcting errors in the previously released versions. Correcting errors afterwards can be acceptable for some systems – but definitely not for all. Systems where errors might lead to security breaches, and systems that are intended to work offline for long periods of time are examples where correcting errors afterwards can be related to extremely high expenses. One tool capable of performing this degree of verification is the ECDAR tool.

The purpose of this project is to contribute to the ECDAR multi-project, to improve upon its functionality. This is accomplished in a larger development environment with multiple student groups, where subprojects must be coordinated for a common solution. The student groups have been utilizing the agile framework Scrum to manage their software development. Furthermore, to optimize cross-group communication, the larger development environment used Scrum of Scrums.

The aim of this group was to further enhance the graphical user interface, while the other groups primarily focused on the Reveaal engine.

The frontend is written in Java with JavaFX to develop the user interface. The backend and frontend communicate with each other using gRPC, which allows them to share a common repository of Protocol Buffers that define the serialization format for transferring data between the two.

The result of this project was the implementation of simulation, reachability check, and other minor fixes.

Number of pages: 99

Date of completion: February 18, 2026

The content of the report is freely available, but publication (with source reference) may only take place in agreement with the authors.

Preface

Citation style

The IEEE citation style will be used to cite the sources in this report. The citations are elaborated on in the bibliography.

Acknowledgements

We would like to thank the other student groups for the great collaboration on this semester. We would also like to thank our supervisor, Martijn Goorden, for thorough feedback. Finally, we send a special thanks to Niels Frederik Sinding Vistisen for his readiness throughout the project.

Target audience

The report is intended for professionals performing model checking in tools similar to ECDAR, and for the future students to get some insights and foundation for their future work on ECDAR.

Authors:

Andreas Paludan

Casper Zacho Roskær

Emilie Sonne Steinmann

Ibrahim Alaiddin Abu Rached

Julie H. Bengtsson

Mikkel Dolmer

Reading Instructions

This report covers the work performed in a 5th semester student project at Aalborg University. The developed product is a contribution to the existing ECDAR solution, which several student groups have been putting a coordinated effort into throughout the project period.

To manage the project work, the agile software development framework Scrum has been enforced. To reflect this, the report is structured in several Sprints. The report follows the structure of the Sprints, which consists of Sprint Planning, an implementation phase, a Sprint Review, and a Sprint Retrospective. Lastly, the report will conclude with a discussion, a conclusion, and a chapter regarding future work, which will serve as a foundation for future students working on the same project.

Throughout the report, sections will appear that have been written in collaboration with the other student groups working on this project. This is marked with a disclaimer in the beginning of the concerned chapters.

Contents

1 | Introduction to ECDAR

The following section and its subsections are partly written in collaboration with the other ECDAR project groups.

This chapter serves as an overview of the different concepts of ECDAR to provide a basic understanding of what ECDAR is, as well as a brief presentation of the theory behind ECDAR and its architecture.

ECDAR stands for Environment for Compositional Design and Analysis of Real Time Systems. ECDAR is a graphical tool to model and analyze real-time systems using timed input/output automata. This process is called model checking. The ECDAR website can be found on www.ecdar.net/.

Model checking is a method of verifying that models, in the form of finite-state automata, fulfill specified criteria [1]. An automaton is defined as one or more components. As the model grows in size, it becomes more complex to verify by hand. The immense complexity of modern systems dictates that the verification must be computer-aided to be feasible [1]. ECDAR is one such tool.

Imagine that you are working for a space agency and your mission is to send a satellite into space. You have spent several months testing the satellite to make sure it is correct and everything is working as intended. All tests have passed for every edge case that your engineers could think of. The satellite is ready to be launched. However, the day after launch, a major problem in the system is found. It needs to be fixed if the satellite is to have any use, but doing so might be extremely expensive or outright impossible, as the satellite is already in orbit.

It is important to model check a system to ensure that it behaves as intended. In the above example with the faulty satellite, there are several interdependent components, which ECDAR is able to take into account. If one of these components fails or is blocked, then it may make the entire satellite faulty. By providing a tool that can be reliable at testing if a new system is true to a given specification, we can not only ensure correctness

but also efficiency in comparison to doing the check by hand.

Model checking as a tool can try to guide the user as much as possible in accordance with the specification, but in the end, the user can still make a wrong specification. Therefore, model checking can only ensure that the system works in the real world if the specifications provided to the system are correct.

As mentioned, the models that we are checking with this tool, are represented as automata – more specifically as timed input/output automata. Therefore, we will first give an introduction to the relevant terminology regarding this. Then later in this chapter, we will introduce ECDAR through presentations of its architecture and current state.

1.1 Timed Input/Output Automata

Timed Input/Output Automata, abbreviated TIOA, are basic automata with two extensions. One of them is the notion of time. Time is tracked using clock variables, which can be used to make constraints in the system on its locations or edges. The second is having actions classified as input or output actions. Input actions are triggered from the outside, and output actions are triggered from within the automaton [2].

The formal definition of a TIOA is a tuple $(Loc, I_0, Act, Clk, E, Inv)$ [2], where:

- Loc is a finite set of locations,
- I_0 is the initial location, so $I_0 \in Loc$,
- Act is a finite set of actions partitioned into inputs (Act_i) and outputs (Act_o),
- Clk is a finite set of clocks,
- $E \subseteq Loc \times Act \times \mathcal{B}(Clk) \times \mathcal{P}(Clk) \times Loc$ is a set of edges, where $\mathcal{B}(Clk)$ is the set of guards over the operations in the set $\{<, \leq, >, \geq\}$ and $\mathcal{P}(Clk)$ is the powerset of Clk, used to update clock values,
- $Inv : Loc \mapsto \mathcal{B}(Clk)$ is a location invariant function.

Here, the guard is a constraint on the edges, and the invariant functions returns the constraints for the locations. Additionally, the clock values can be updated, i.e., their values can be set to zero, when taking a transition. Updates are defined on the edges.

The semantic representation of a TIOA is called a *Timed Input/Output Transition System*, TIOTS, and is used to analyze TIOAs. A TIOA is a finite representation of a TIOTS. A TIOTS is a tuple $(Q, q_0, Act, \rightarrow)$ [2], where:

- Q is a possibly infinite set of states. A state is a location and the values of the clocks [3],
- q_0 is the initial location, so $q_0 \in Q$,
- Act is a finite set of actions partitioned into inputs (Act_i) and outputs (Act_o),
- $\rightarrow \subseteq Q \times (Act \cup \mathbb{R}_{\geq 0}) \times Q$ is a transition relation.

If all states are input-enabled, the TIOTS is a *specification*, which means that each state in the corresponding TIOA are be able to receive an input [2].

1.2 Architecture

ECDAR consists of four major parts as seen in Figure 1.1. There are the two engines, Reveaal and j-ECDAR, as well as the graphical user interface (GUI) of ECDAR, and a test framework. The GUI and the engines communicate through gRPC (Google Remote Procedure Call) which uses Protocol Buffers (Protobuf) to serialize data structured as messages [4]. The advantage of this being the ability to work across languages and across platforms [4, 5], simplifying the integration process and making it easy to use.

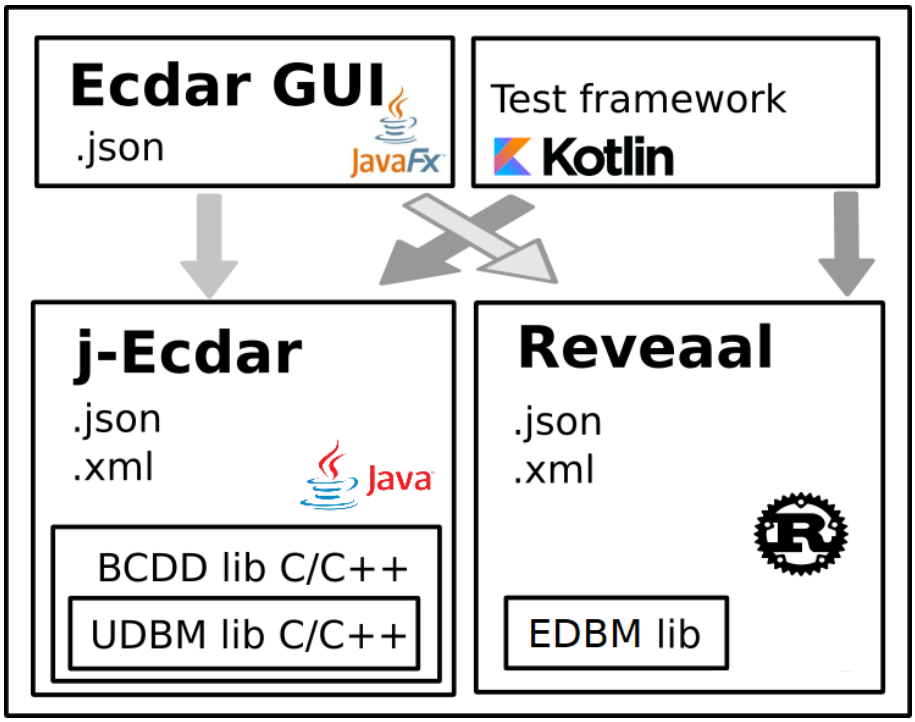


Figure 1.1. The architecture of ECDAR visualized [6].

The GUI is written in JavaFX which is a graphics and media library for Java [6]. The GUI provides the tools which enable the user to model their real time systems.

ECDAR runs on two verification engines: j-ECDAR and Reveaal. j-ECDAR is written in Java and Reveaal is written in Rust. Recently, the libraries that Reveaal uses have been rewritten in Rust from C/C++, with the intention of implementing multithreading. The purpose of having two different engines is to make the whole platform more reliable. With two engines, their results can be compared which will help ensure correctness in both engines. The main priorities for the j-ECDAR engine are readability and correctness, and, as stated on ECDAR's homepage: "no effort is put into optimizing the code for speed" [6]. It should be noted here that j-ECDAR will not be worked on for this semester, as all focus is allotted to the Reveaal engine. In contrast to j-ECDAR, the Reveaal engine is intended to be fast and parallelizable.

ECDAR makes use of a testing framework written in Kotlin. The testing framework uses a collection of test cases to test both of the engines. The testing framework is vital to perform conformance testing between j-ECDAR and Reveaal as well as automated performance testing and hand-designed test cases.

1.3 Current State of ECDAR

This section describes the state of the ECDAR GUI and the Reveaal engine, when we took over the solution in the beginning of the product period. This was the offset for our work.

1.3.1 State of the GUI

The GUI is split into three vertical panes, as seen in Figure 1.2. The different components in the project are listed in the project pane to the left, the currently selected component is shown in the editor pane in the middle, and the queries are found in the query pane to the right. Through the editor pane, components can be created, modified, and deleted. Projects are saved in their own folders with a JSON file for each component, allowing the components to be used for queries in the engines of ECDAR. The GUI also has a top bar, from where it is possible to open a project, hide panes, choose engine options, and more. Additionally, the GUI has a bar at the bottom of the view. Here, errors and warnings are supposed to be displayed.

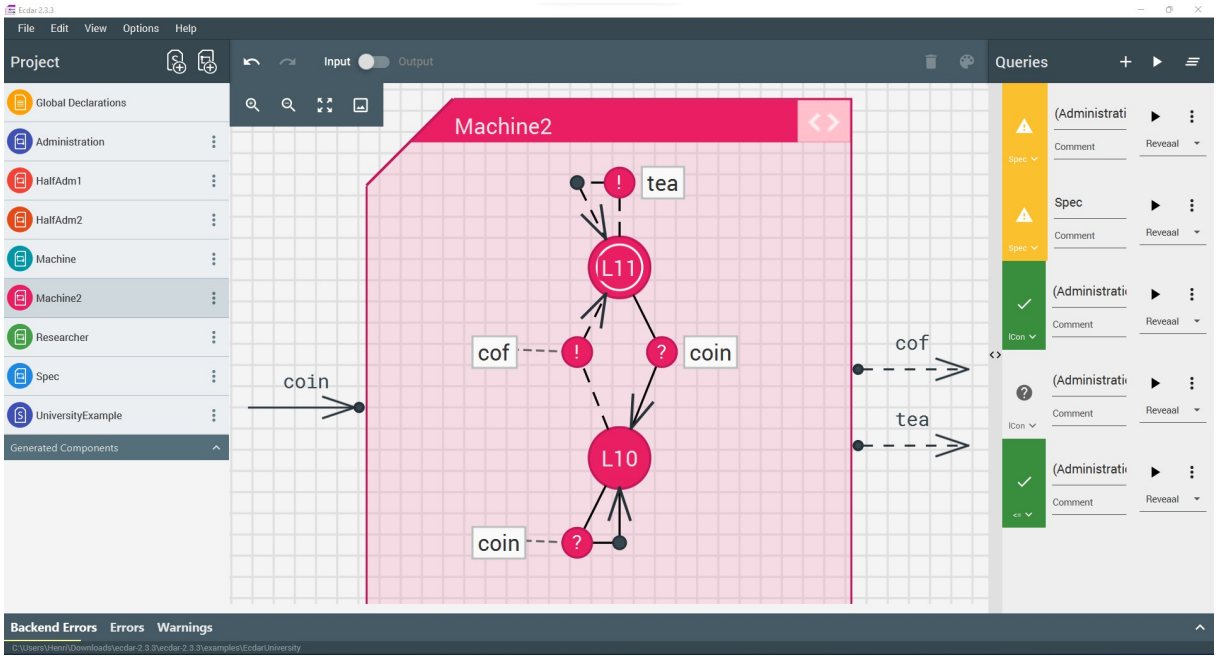


Figure 1.2. A screenshot of the ECDAR GUI of the latest version at the start of the project period (ECDAR 2.3.3 released on 2022-09-08) [6].

In the example above, `Machine2` is selected and presented in the editor pane. `Machine2` is a model of a coffee machine. The accepted input is shown on the left side of the component as a solid arrow pointing towards the component, here the input is `coin`. On the right side the output from the coffee machine, namely `cof` and `tea`, is illustrated with dashed arrows pointing away from the component. Inside the component, the model is composed of locations in the shape of big circles, and transitions marked by arrows between the locations. The initial location, `L11`, is marked by a white circle around the text. The two different types of transitions (input and output) are distinguishable through the type of line (solid or dashed), and through the symbols in the small circles on the arrows. The exclamation mark indicates an output, e.g., tea on the transition from location `L11` to itself at the top of the model. A question mark, on the other hand, indicates an input. An example of that is the transition from `L11` to `L10`.

Apart from the mentioned features, it is also possible to add guards and updates, in addition to invariants, by right-clicking the transitions and locations, respectively. An example of a guard is provided in Figure 1.3 as a circle containing the less-than symbol with an associated text box in which the actual guard is added and can be edited. An update is represented in a similar manner, but with the equals symbol in the small circle. This is shown in Figure 1.3 on the transition from `L5` to `L4`.

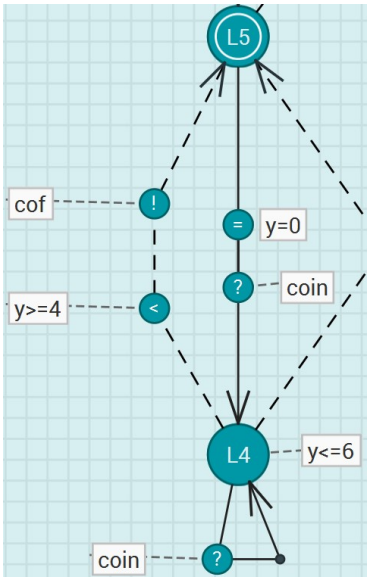


Figure 1.3. A screenshot of a model containing guards, updates, and invariants.

An example of an invariant can be seen in Figure 1.3 as a text box with a dashed line connecting it to location `L4`. The invariant is added in a manner similar to that of the guards and updates.

As mentioned earlier, queries can be executed from the query pane to the right of the GUI, removing the necessity of using the command line interface for query execution. Here, all created queries can be run at once or individually. A query consists of a query request specified in the text-field, a query type chosen using the drop-down underneath the icon, and the selection of a backend engine for execution. The drop-down menu contains a preset of query types, but not all options are currently supported with functionality that being e.g. reachability.

The run of a query can result in one of three things: A green box that indicates a successful run, a yellow box that indicates that the query could not be parsed by the backend or other issues occurring, or a red box that indicates that the query is unsuccessful, e.g., some check in the engine fails.

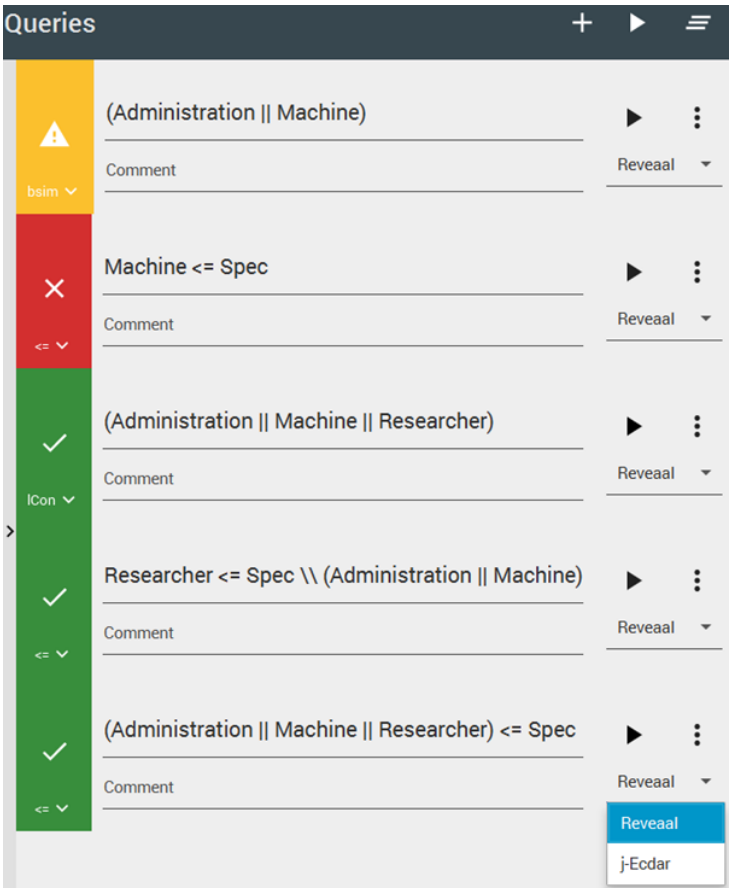


Figure 1.4. A screenshot of the query pane.

Figure 1.4 shows five different queries. The result of the first query in the query pane is an error, which means the backend could not handle the query. Underneath this query, the "Machine<=Spec" query is performed. The result of this query is unsuccessful. The three remaining queries are all successful and without errors. A limited amount of information about the results of the queries can be obtained by hovering the mouse over the colored icons.

1.3.2 State of Reveaal

Reveaal is an engine designed to check the correctness of a real-time compositional system made by a model designer. It is able to test a variety of different types of queries, such as refinement and consistency, making it a powerful tool for ensuring the correctness of systems. Reveaal uses gRPC, allowing for quick and easy communication between model designers' choice of frontend and the engine itself. ECDAR does, however, currently only have a single GUI. The query handler within Reveaal is responsible for recognizing the type of query it receives, and calling the correct functions to perform the query. Upon completion of the query, Reveaal will return either a success or an error message.

Though Reveaal provides basic functionalities (outlined in [2]), its error handling and user-friendliness leave much to be desired. Errors are not communicated to a user as the Reveaal engine just panics, leading to a generic error message in the GUI, which does not provide any information regarding the violating action or the unreachable part of the model. Consequently, users are not provided with an optimal experience.

1.4 Plan for ECDAR

During the project period, the state of ECDAR should evolve to encompass more features, and by those means, be more complete. Each team was assigned a part of ECDAR to work on. During the multi-project, the following features were laid out as tasks that would be worked on and hopefully completed. It was the goal that by the end of the semester Reveaal would be able to execute simulation, have multi-threading implemented, check if certain locations are reachable, be able to reduce clocks, and communicate the causes of failures to the GUI. By implementing these features, Reveaal would have many more capabilities than before, and be closer to the end goal of ECDAR. Meanwhile, the GUI team was tasked with developing the necessary representations for the engine features, such that the GUI is up to date with the engine. If all these goals were achieved, ECDAR would, on a feature level, be much more functional than it was in ahead of the semester.

End Goal

Student developer Sebastian Lund describes the end goal for ECDAR as:

> The end goal of ECDAR is to be a full Environment of Compositional Design and Analysis of Real time systems. For this to be realized it must be seamless to switch between the tasks of designing specifications and testing specifications. With every user design change, the tool should continuously test the specification to provide immediate user feedback and suggestions. When any query fails the user should be able to debug the execution with the simulation tool and easily modify the specification. The user should have the freedom to design specifications with the option of boolean and integer variables. The design freedoms and continuous testing make the efficiency of the verification engine essential for the responsiveness of the tool. Once users are satisfied with their specification design, ECDAR should be able to generate a set of implementations for the user to choose from and generate code for the implementation in a contemporary programming language.

Furthermore, he also gave a non-exhaustive list of future features for ECDAR:

- Support for the step-by-step simulation of every query type
- Support for boolean and integer variables
- Continuous testing on specification changes
- Automated model change suggestions to fix failing queries
- Automatic generation of implementations that satisfy a specification
- Automatic code generation from implementation specifications
- Tools for translating code interfaces to ECDAR specifications
- Automation of specification simplification by generating smaller bisimilar (same behaviour) specifications
- Development of theoretical real time abstractions for exponential reductions in query runtime and memory

At the end of this project, we hope to have furthered ECDAR towards its aim, providing new features and fixes that would improve both performance and user experience. By possibly fulfilling one or more features, ECDAR would stand more complete than when we began working on it.

2 | Organization

The following section and its subsections are partly written in collaboration with the other ECDAR project groups.

This semester project focused on multi-project collaboration, which means that several project groups cooperated on the same project. In total, six project groups consisting of about six members each collaborated to extend ECDAR further. Five of these groups focused on further development of the Reveaal engine, while the last group focused on developing the GUI of ECDAR. To orchestrate the collaboration, we established an organizational framework, in which the work could take place. This chapter serves to document our organizational thoughts from the beginning of the project work.

As one of the purposes for this project was to perform agile software development, an introduction to this concept and the Scrum framework will first be given. In the later sections of this chapter, it is then described, how Scrum was intended to be implemented and how the work was organized – both on the higher level coordinating the joint effort of the six groups, as well as on the lower level within our own group. Naturally, this last section (Section 2.3) is *not* written in collaboration with the other teams, as it only applies to our group.

2.1 Software Development Methodologies

This section gives an introduction to software development methodologies in general. As part of this project is to learn how to work in an agile manner, special emphasis is put on *agile* software development. Supporting this, Scrum will be introduced as a specific agile framework.

2.1.1 Agile Software Development

Before agile development, project management relied heavily on the waterfall method. The waterfall method entails planning everything before proceeding with development in stages. Every stage is completed entirely before the next begins. The method can be useful when both the problem and implementation are clearly defined and unlikely to change during the process. The problem, though, is that it has no mechanism for handling uncertainty [7].

Because we were going to work with a codebase, languages, and techniques that were completely new to most of us, it seemed difficult for us to estimate workloads and plan ahead; neither the problem nor the implementation was well-defined enough for us to apply the waterfall method. Additionally, the requirements for our further development of ECDAR did not seem clear, so this imposed a high risk of change during development, which the waterfall method could not handle. Based on these arguments, it was very clear that the waterfall method would not be suitable for this project, even if we had wanted to follow that method.

An alternative to the waterfall method is the agile development practices, which arose from a need for a development method that would solve the issues posed by the waterfall method. Most importantly, it would lessen the cost of making changes to the specification late in the development process [8]. The agile development practices must support the values outlined in the Agile Manifesto [9] from 2001. The first statement of the manifesto is: "Individuals and interactions over processes and tools" [9], which means that while we decide *who* will be solving a given problem, we aim not to restrict them in terms of processes they must adhere to, or tools, they would be forced to use. Other values state, e.g., that functioning code is more important than documentation, and that customer collaboration is of higher value than excessive contract re-writing, as a better understanding of the problem throughout the process reveals hidden needs.

As the priorities for the development of the ECDAR project were not well-defined and might change throughout the project period, working in an agile manner seemed suitable. This, however, required consistent communication with the "customers", which we had good access to in the shape of members of the professional ECDAR team. In agile development, the focus is more on delivering the software that the user wants than it is on planning ahead, and we hoped that this would allow us to develop good software without stressing about irrelevant uncertainties. One thing each group had to take into consideration, however, was the lack of focus on documentation, which was contradictory to the nature of writing a project report documenting all Increments.

In order to perform agile software development, one can choose to follow a specific framework. We chose to use the Scrum framework, which is introduced in the next section.

2.1.2 Scrum

Scrum is an agile framework that developers use to ensure an agile development process where *Commitment*, *Focus*, *Openness*, *Respect*, and *Courage* are valued and practiced [10]. Scrum defines different roles, artifacts, and events, which will be introduced in a theoretical manner in this section. Then, in Section 2.2 and Section 2.3, the usage of Scrum for this particular project will be made clear.

Roles

The Scrum framework describes a team that consists of a *Scrum Master*, a *Product Owner* (PO), and *Developers*. These roles, which are described in more detail below, are all equally vital, and are not positioned in a hierarchy. Instead, everyone on the team is responsible for playing their part in the development and ensuring that the software is delivered. Scrum is based on having a small and nimble team that is still large enough to make significant progress during each work period. These teams should typically be less than 10 people [10].

The Product Owner is responsible for maximizing the value created by the Scrum teams. It is also the PO's responsibility to manage the *Product Backlog* as well as developing and communicating the *Product Goal* [10].

The Scrum Master is responsible for making sure the team is focused on reaching the goal and maintaining an environment where the Scrum values are upheld. This includes ensuring that the team follows the Scrum framework, removing obstacles for the team as well as facilitating the different Scrum events. The Scrum master is not a manager, but a member of the team and is on equal terms with the rest of the team [10].

The Developers make up the rest of the team members. The developers are accountable for creating usable *Increments* during the Sprints. An Increment is a tangible step towards the Product Goal, which meets the Definition of Done (DoD). Developers take part in creating the *Sprint Backlog*, ensuring that Increments adhere to the DoD and adapting their daily plan in accordance with the current Sprint Goal [10].

Scrum Artifacts

The Product Backlog is one of the three artifacts in Scrum, where the progress of the *Product Backlog* is compared to the Product Goal. The Product Goal is a description of a future state for the product, typically its final form. The Product Backlog is an ordered list that indicates which components of the product need to be improved as well as which features need to be made [10].

The Sprint Backlog is a subset of items from the Product Backlog that the Scrum team has selected for the particular Sprint. The Sprint Backlog is created by and for the developers in the Scrum teams. The Sprint Backlog is initially created during the *Sprint Planning* event. The Sprint Backlog can be updated during the Sprint if, e.g., the Scrum team gains new knowledge about the problem [10].

The Increment is the third and last artifact in Scrum, and it is a concrete step towards the Product Goal. It is possible that multiple Increments are created during a Sprint [10], as an Increment is when a feature from the Product Backlog meets the DoD. The DoD is a shared definition for when an Increment meets the quality standards of the product and thereby describes when an Increment is created. When multiple Scrum teams work together, a mutual DoD must be defined [10].

Scrum Events

Scrum contains five events which are designed to enable transparency and to create opportunities for course correction [10].

The Sprint is an event with a fixed length, typically a month or less. During the Sprint, items from the Sprint Backlog are turned into Increments by the developers, thereby creating value [10]. The remaining events all happen during a Sprint.

Sprint Planning is the process of planning the upcoming Sprint. The plan should specify three things. Firstly, the plan should address why the Sprint is valuable to the development of the product. Secondly, it should address which items from the Product Backlog will be added to the Sprint Backlog. Finally, the team should discuss how they will approach each item from the Sprint Backlog and ensure that it will meet the DoD [10].

Daily Scrum is a short meeting that is held every day. Here, it is briefly discussed what each participant did the previous day, and what they plan to work on this day. Lastly, if the participants have any challenges these will be discussed as well. The goal of this event is to improve the group members' communication, to help identify and solve challenges,

and to create an opportunity for the team to share knowledge. If needed, the Daily Scrum is also used to adapt or re-plan the work of the ongoing Sprint [10].

Sprint Review is an event held at the end of a Sprint. The purpose is to inspect the outcome of the Sprint and to adjust the Product Backlog accordingly. PO, Scrum Master, Developers, and Customers can participate in the Sprint Review [10].

Sprint Retrospective is the very last event during a Sprint. In a Sprint Retrospective, the team discusses how the processes and interactions during the Sprint helped or hindered the development process. Then, the team will discuss how it can improve its effectiveness and the quality of the outcome in the Sprint to come. The retrospective differs from the review in its focus; the retrospective focuses on the work process, while the review is concerned with the product [10].

2.2 The ECDAR-SW5 Organization

To orchestrate the collaboration between all Scrum groups, it was essential to organize how communication would take place, as well as how we would distribute and evaluate work.

Each of the project groups was themselves a Scrum team. As Scrum is dependent on the teams being small, the organization must use an agile scaling framework. Such a framework ensures distribution of the workload between the teams and that the output from each team can be integrated.

As an organization, the ECDAR-SW5 organization found it crucial to consider its needs before picking a framework to implement. Due to the requirements from Aalborg University as well as what is preferred within the six individual groups, each group must be self-managed and free to decide how they work internally. From these requirements, we decided that Scrum of Scrums was a good way of scaling our organization. Scrum of Scrums was chosen, as Scrum was already known and preferred. The Scrum of Scrums framework allowed each team to still work as one team, while still providing teamwork across teams. Other possible solutions exist, we might have chosen to use eXtreme Programming, or any other agile framework. The choice of Scrum of Scrums, was made due to it being the most well known, and well understood method.

2.2.1 Scrum of Scrums

As described in the previous section, we have chosen to use Scrum of Scrums (SoS). SoS is the implementation of a Scrum framework on top of multiple groups working with the Scrum method, essentially creating a high-level Scrum team. Each Scrum team should therefore choose a member to attend the SoS meetings.

The Daily Scrum event is only used by each individual group, and not by the overarching SoS. This was decided as each group works on independent features and thus a Daily Scrum between the groups is unnecessary for the given project.

The key events in SoS are, like they are in Scrum, the Sprint Planning, Sprint Review, and Sprint Retrospective. These events are handled by representatives of each group, ensuring that the collaborative processes between groups are agreed upon.

To allow for short and concise SoS meetings, each group should have internal Sprint Planning and Review and Retrospective meetings after the Sprint Planning and before the Review and Retrospective. This ensures that the representative is ready for the main SoS meeting. This process is also illustrated in Figure 2.1. As seen in the figure, the length of each Sprint was decided to be two weeks, and we would hold the Sprint Planning on the first day of the Sprint, and the Sprint Review and Retrospective on the last day of the Sprint.

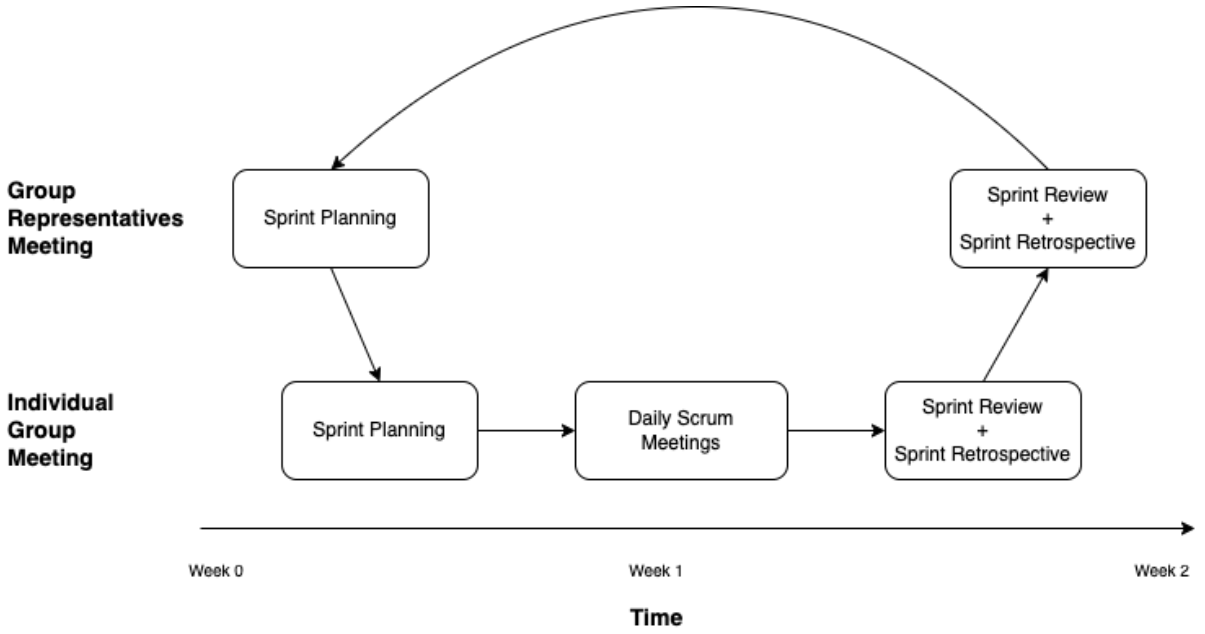


Figure 2.1. A figure depicting the events in a Sprint, divided between the big SoS group and the smaller Scrum groups.

As mentioned, the multi-project focuses on both the GUI and Reveaal engine. Because of

this dual focus, we have two POs: one for the Reveaal teams and one for the GUI team. The project's different POs will help uphold the Product Backlog and update it if any changes appear. These changes are discussed during the SoS meetings, where the POs can also be present.

2.2.2 The Mutual Definition of Done

A DoD had been agreed on to define when a feature is considered done and when an Increment is created. A DoD had also been necessary to better facilitate multiple Scrum teams working on the same codebase. The DoD was defined and described at the first Scrum of Scrums meeting.

We decided that every pull request to the GUI- and Reveaal-repositories' main branches should be reviewed internally in the team as a draft before it is sent out for an external review. The external review consists of four different checks:

- Ensure the new functionality described is implemented and works.
- Run a benchmark on the main- as well as the new functionalities branch (specific for Reveaal).
- Check if the new code is properly documented. The I/O and panic criteria should be documented (specific for Reveaal) as well as what it does if functions are public.
- All tests should pass and if there is a new functionality it should be tested. It is up to the reviewers to judge if the tests are good enough.

Secondly, the new functionality should be approved by two external reviewers.

If none of these areas were found lacking, and both reviewers approved of the changes, it was considered done.

It is worth noting that even though we did not make a formal definition for anything other than the code, a review process for the collaborative part of the report existed. The review process was threefold. First, each Scrum team separately reviewed the report. Then each team was assigned an area of responsibility, which was corrected and proofread. Lastly, the original person who commented on the now corrected part, either accepted the change, or contacted the team who made the corrections. After this process, the collaborative writing was considered done.

2.2.3 Shared Tools for Collaboration

All Scrum groups shared a common toolbox. This toolbox contained tools that helped the groups cooperate with each other. The primary tool was GitHub, and it was used for version management.

The groups were not directly altering the original "upstream" repositories of ECDAR. Instead, the groups forked the repositories of the original ECDAR project. Forking repositories means that new repositories were created that shared the code and visibility settings with the upstream repositories of the original project [11]. This strategy was used, since the changes to ECDAR that the groups were proposing had to be approved by the developers that are in charge of the ECDAR project, and it was therefore not desired that the groups' proposals were merged directly into the upstream repositories.

At the end of the semester, the developers in charge will look at the proposed Increments, and it is up to them to decide what should be kept for the upstream repositories. It should therefore be noted, that not everything implemented by the groups during this semester will necessarily become a part of ECDAR.

The Reveaal groups had one repository and so did the GUI group. Each group then created their own branches, minimizing the risk of groups getting in the way of each other's work. Because of this, a branching strategy was made during the first Scrum of Scrums meeting. For each Increment, a branch was made from the main branch on the specific repository. When an Increment met the DoD, the branch could be merged back into the main branch.

To keep track of Increments and both the Product- and the Sprint Backlog, a project board for each repository was created on GitHub. These boards held the information of every different Increment that the individual groups worked on.

Another tool in the toolbox was Discord. A Discord server was created and used for communication between the groups. The server contained different text channels that were used for different purposes, e.g., a channel for collaborating on writing the joint sections in the report, and a channel for setting up meetings.

OneDrive was another tool in the common toolbox. For every meeting in the Scrum of Scrums, an agenda was created and distributed between the groups through OneDrive. OneDrive was also used for sharing documents, notes, etc.

The joint writing was done in Overleaf and later uploaded to GitHub in a repository. Using Overleaf made it possible to write together in real-time and later insert the sections

into the individual groups' reports since all the groups write their reports in Overleaf.

2.3 Internal Organization

Internally in our project group, we also implemented the Scrum framework. We worked in six Sprints, each with a duration of two weeks, in alignment with the structure described in Figure 2.1. To guide the work within the Scrum framework, a group member was chosen as Scrum Master, and another group member was chosen as the representative, who would participate in the SoS meetings, as explained in Subsection 2.2.1.

Regarding the direction of the product development, a member of the official ECDAR development group agreed to take on the role of the PO. This was a great advantage, as it resulted in a PO who was very involved with the product, and who had the level of authority and understanding required to ensure that the delivered Increments were valuable to the overall product. The PO could be contacted within normal work hours and had an office located close to our facilities, which was ideal in terms of the availability of the PO. Even with the presence of a PO, we still wished to have some influence on the types of contributions, we could make. Therefore we launched the project work by first setting up a Product Backlog in GitHub, as seen in Figure 2.2.

	Title	Assignees	Reviewers	Status	Sprint	Linked pull requ...	Interest	Difficulty
	No Status 11							
9	Autocorrect the symbol for refinement (>= to ≥) #12	Emilie...					2	1
10	Beautify selfloop #13						2	2
11	Display error messages when running operations #6	APalu...					3	2
12	Update to run with Java 17 #10						1	3
13	Display error message (JAVA HOME) when trying to launch program #8	APalu...					2	3
14	Simulation and debugging #5						2	4
15	Visual improvements to "Spider's web" #7						1	4
16	Performance updates for view rendering #9						3	4
17	Bundle GUI med Java (rigtig version) så man ikke behøver at have det insta #14							
18	It should not be possible to run a query when pressing enter, and no query #19							
19	Width of query symbol/icon is not same #21							

Figure 2.2. A screenshot of the Product Backlog which is implemented using GitHub Project boards [12].

The Product Backlog contained a set of issues that were partly suggested by the PO and partly arose from our interests and experiences with using ECDAR. Each issue was described by providing a description of the experienced problem/desired feature along with a DoD. The Product Backlog was reviewed with the PO to ensure relevancy and discuss priority of the different issues. The issues were categorized by level of difficulty

and level of interest, to facilitate the selection of issues for a Sprint. Later, they were also given a priority from the PO's perspective.

The PO also agreed to participate in relevant meetings throughout the Sprints. As mentioned in Subsection 2.1.2, there are four different types of meetings being held within a Sprint: Sprint Planning, Daily Scrum, Sprint Review, and Sprint Retrospective. We as developers participated in all meetings, and the PO also attended the Sprint Planning and Sprint Review. In alignment with the meetings in the SoS meetings, we performed Sprint Planning on the first Monday in the two-week period of a Sprint, and Sprint Review and Sprint Retrospective on the last Friday of the period, when possible. A sketch of the structure is shown in Figure 2.3.

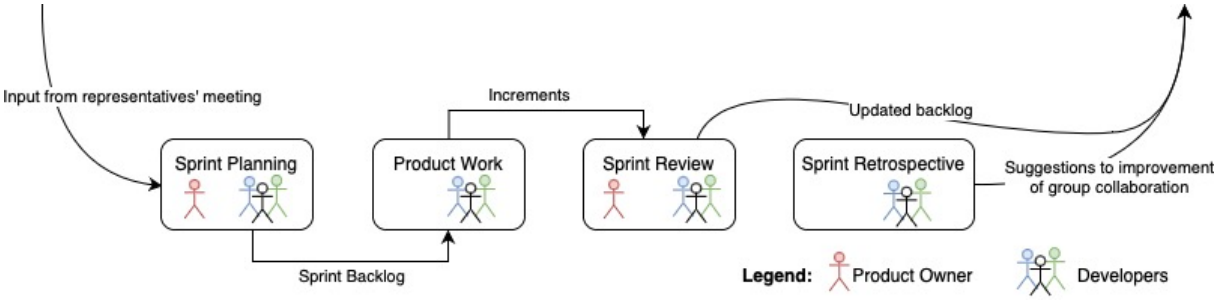


Figure 2.3. Sketch of Sprint flow within our group with input/output and participants specified.

The Sprint Planning took the output of the SoS Sprint Planning into consideration. The output of the meeting was a Sprint Backlog, which we implemented in a GitHub Project with links to the Product Backlog, but with a much higher detail level, combined with an overall Sprint Goal. A snapshot of the GitHub Project board can be seen in Figure 2.4.

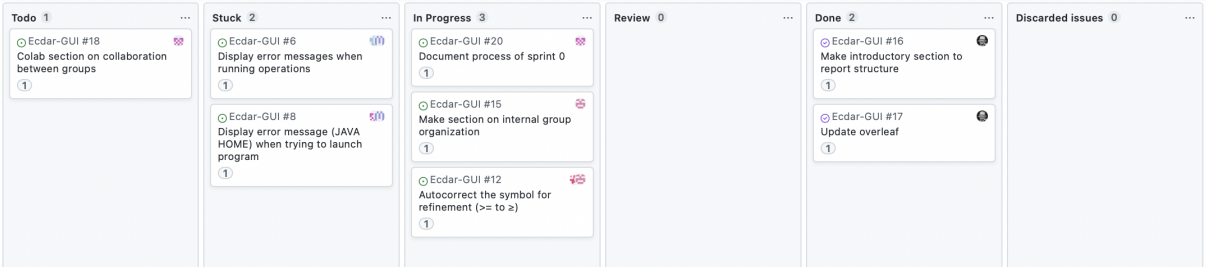


Figure 2.4. A screenshot of the Sprint Backlog, excluding the Sprint Goal which is implemented using GitHub Boards

After the Sprint Planning, the work on the product began. For this project, we had approximately eight workdays in each Sprint, within which we also had lectures to attend. Before we started working for the day, we conducted a Daily Scrum, where all group members described what they had been working on since the last meeting, and what they would work on today. There might also be other issues that were briefly discussed at this

meeting, as we were all gathered there. To ensure that these meetings were kept concise, all participants had to stand up for the meetings.

To assess the developed Increments, we had a Sprint Review on the last Friday of the Sprint, as mentioned. The PO as well as the developers participated in this meeting, as all must agree on the status of the issues that had been the focus of the Sprint. The Increments presented at this meeting were assessed by the developers beforehand to ensure that the lived up to their three-part DoD. The DoD consisted of a set of specifications for the given issue, i.e., what functionality this issue was requesting, combined with both an internal and an external set of guidelines to when an Increment was accepted (regarding internal reviews, testing, etc.) by our group and by the ECDAR project groups in unison. The issue-specific DoDs were described in each issue. The general DoD we developed for internal use in this group is as follows:

- The code contributions must successfully have undergone high-quality testing.
- The code contributions must have been reviewed by two group members.
- The code contributions must be readable, i.e., using descriptive variable names and a logical structure.

This DoD is an addition to the common DoD described in Subsection 2.2.2.

At the Sprint Review, the PO confirmed whether the Increment satisfied the requirements from their perspective. The Product Backlog would also be revisited and issues completed within the Sprint were marked accordingly.

Finally, the developers conducted the Sprint Retrospective with the purpose of evaluating the effectiveness of the work done within the Sprint. Discussions were based on any experienced problems and suggestions, the developers might have had, and the outcome was in the shape of suggestions to change that can be implemented within the next Sprint. To ensure that the outcome of this meeting was taken into consideration and tested in the next Sprint, they were formalized as issues in the Sprint Backlog of the next Sprint. The Sprint Retrospective concluded the Sprint and served as an offset to the next Sprint.

3 | Our Field of Work

This report will focus on the ECDAR GUI, which is the frontend layer for both the Reveaal and j-ECDAR backend engines. The main purpose of the ECDAR GUI is to provide a frontend system which can streamline the user interface for both engines. At the time of writing, the roadmap for the GUI is as follows:

- **Modelling of components**
- Query execution using engines
  - **Refinement checks**
  - **Local consistency**
  - New component generation
  - Reachability
  - Specification
- Simulation
- Visualization of Zones

Of these *Modelling of components*, *Refinement checks* and *Local consistency* have been implemented. We aim to implement a graphical representation of both *Simulation* and *Reachability*. However as neither of these project issues has been implemented in the backend, it is necessary to cooperate with the backend teams in order to reach our goal.

3.1 Simulation

Two features of ECDAR that had not been implemented were simulation and reachability. As our group has been involved in the implementation of both, a general introduction to both concepts is given here.

In the context of ECDAR, simulation refers to the user's ability to investigate the components and the traces that can be followed within each component. This should be implemented in a manner, where the user is able to start a simulation, which will then open in a new view. Here, the components specified in the query to start the

simulation must be shown. The simulation starts in the initial state, and the locations of the current state are highlighted at all times, so at the beginning, the initial locations will be highlighted. Furthermore, the transitions that can be taken from the current state are also highlighted.

It should be possible for the user to investigate the components by either performing a possible transition, which will lead to a new state, or by selecting a clock that is incremented until a transition is forced to happen, and a change occurs in the set of locations in the current state. A backend team has been implementing the computations required to determine possible transitions and to compute the new state based on taken transitions or clock Increments. Our tasks have thus been to improve the visual layout of the simulation tool, to which some preparations have already been made, as well as set up the GUI part of the communication with the backend, i.e., to send requests and to receive and present results from the backend.

3.2 Reachability

Reachability, on the other hand, refers to the possibility of performing a check of whether a location is reachable from a given state. To test this, the user should be able to right click a location in the simulation view to invoke the reachability check. The reachability check should return a boolean value that represents whether the location can be reached or not. If the location can indeed be reached, it should also return a path to the location from the start state defined in the request.

A backend team has been implementing the logic to perform the actual check and calculate a path. The task for our group has then been to give the user the option of sending the request for a reachability check from either the initial, current, or a user-defined state, to the selected location. The request must comply with the syntax developed by the backend. We have also been responsible for visualizing the response in the simulation view by showing the result and highlighting the path in the components if the location was reachable.

In conclusion, the ability to perform simulation and reachability checks would allow the user to investigate the components and their behavior when different edges were taken or clocks increased. Moreover, it would be possible to test the accessibility of different locations from different states.

Therefore, we have been working on implementing the simulation view in the user interface along with adding new queries to support the functionalities of simulation and reachability.

4 | Sprint 1: Onboarding I

Duration: 19/9/2022 - 30/9/2022

4.1 Sprint Planning

Sprint 1 started with the SoS Sprint Planning meeting. At this meeting, each group presented issues found in the current version of ECDAR, which they would like to work on. The issues we identified in this group are seen in Figure 2.2. Together with input from the POs, this prioritization then determined what the groups would be working on in this Sprint. Furthermore, at the SoS meeting a collaborative writing among the groups for the introductory sections of the report was arranged. It was also discussed that this first Sprint would require some time for onboarding, since it takes time to not only understand the purpose and current state of ECDAR, but also to explore how the project can be further developed. Therefore, onboarding was considered part of the Sprint Goal.

As shown in Figure 2.1, the group SoS' Sprint Planning is followed by the Sprint Planning for our own group. Several small issues from the GUI Product Backlog were assigned to the group Sprint, as solving these supported the Sprint Goal about familiarization with the codebase of ECDAR. It also supported our submersion into the principles and work methods of Scrum, as it allowed for spending more time on important discussion about group organization, meeting structure, and cooperation with the PO.

Specifically, we decided that in this Sprint we would try to solve issues #8, #12, and #16. These issues were concerned with displaying error messages and symbols correctly to the user, so they seemed like a good starting point for gaining an understanding as to where data from the backend is received, and how it is shown to the user in the frontend. To solve these issues we would first make sure that all group members were able to run the application. We would then work in pairs and individually in the group room to gain an understanding of different parts of the solution, which we would share with the other

group members. We would reach out to the PO quickly if we got stuck in order to ease the familiarization process.

In conclusion, what made this first Sprint valuable was that our group gained a clear understanding of what the purpose of ECDAR is, how we could further expand ECDAR, and how our group could utilize the principles from Scrum as good as possible.

4.2 Implementation

In this section, we will describe how two of the three selected issues were solved. Issue #8 will not be described further, as it was not possible to replicate the error described in the issue.

4.2.1 Issue #12 – Autocorrect Mathematical Comparison Symbols

Issue description: When writing operations in the ECDAR GUI, the comparison operator symbols "≤" and "≥" are currently displayed naively as "<=" and ">=", if the user types in the two symbols separately. As this is not the correct mathematical notation, it would improve the readability of the GUI, if ECDAR could autocorrect the sets of symbols into the corresponding, correct, mathematical notation.

Definition of Done: The ECDAR GUI autocorrects ">=" to "≥" and "<=" to "≤". When choosing the query type "refinement" in the drop-down for query types, "<=" should be replaced with "≤".

Implementation description: To ensure that "≤" and "≥" will be displayed in the GUI every time that "<=" and ">=" is written in the query text field, a `ChangeListener` has been added to the text field. The `ChangeListener` is then invoked at every change of the text within the text field. When invoked, the `ChangeListener` will check if the string within the text field contains either "<=" or ">=" as a substring. If this is the case the substring will be replaced with its corresponding Unicode, making sure that it will be displayed correctly in the GUI, as seen in Figure 4.1.

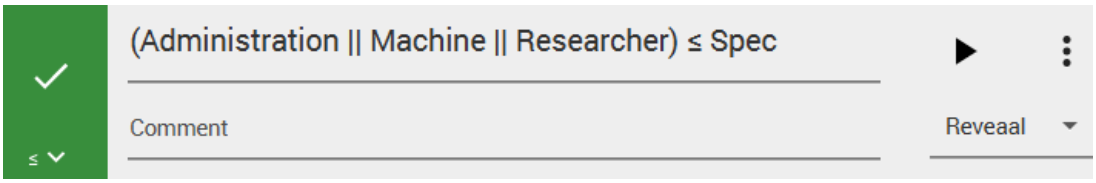


Figure 4.1. Refinement symbol is corrected from "<=" to "≤" when writing a query and when the query type "refinement" is selected, its symbol is also displayed correctly, as seen below the checkmark in the green box to the left.

This, however, led to some problems when sending the query from the GUI to the backend, as the backend currently only accepts the substrings "<=" and ">=" and not the Unicodes of the correct mathematical symbols. To make sure that the backend could receive the queries again, the method called `getQuery()` for getting the information on `Query` objects was modified. In `getQuery()`, the Unicodes of "≤" and "≥" are replaced with the "<=" and ">=" symbols, respectively. A unit test was made to ensure that the `getQuery()` method returned the correct output.

4.2.2 Issue #16 – Show Available Error Information

Issue description: As of now, error and status messages are shown to the user with only a very brief description of the problem or no description at all, as shown in the screenshots in Figure 4.2. This is not very informative for the user and does not at all guide the user in how to proceed in resolving the error. To support the use of the application, this should be improved.

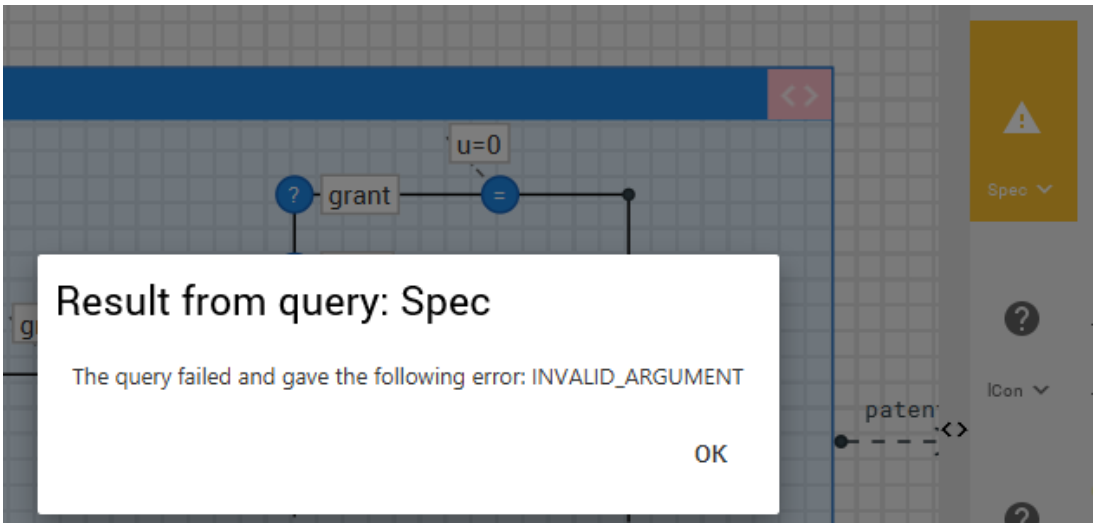


Figure 4.2. An example of a vague error message. This is representative of the general level of detail in error messages.

Definition of Done: When an error occurs, information is displayed about the

specification that caused the error and the state in which it happened, if this can be determined. Additionally, a general suggestion on how to solve this type of error is provided.

Implementation description: When researching the issue, we found that no information about why the error happened was received from the backend. This was in contrast to the information we had received from the PO. However, another student group was working on sending elaborate error messages from the backend, so after discussions with them, we decided to postpone most tasks of this issue to a future sprint. This way the backend team had time to implement the necessary error handling. However, some adjustments have been made to inform the user when a query fails. A method responsible for updating the tooltip was missing a condition for when a query fails, resulting in an empty tooltip. This has now been fixed, and the result can be seen in Figure 4.3.

(a) Before: Tooltip displays no information about the status of the query.

(b) After: Tooltip displays simple information on the status of the failed query.

Figure 4.3. Difference between the tooltip before and after the Increment.

4.3 Sprint Review

The goal for Sprint 1 was to get a good overview of the code for the ECDAR GUI and the Sprint Backlog therefore contained three smaller issues that would help achieve the goal. A secondary goal of the first sprint was to set the structure for the report and for all the remaining Sprints.

One of the issues in the Sprint Backlog was to autocorrect the refinement symbol when writing a query (issue #12), and this issue was solved during the first Sprint, as described in Subsection 4.2.1.

The two other issues assigned to the Sprint Backlog were impossible to solve in this Sprint due to the lack of information delivered by the backend.

One of the unsolvable issues was issue #16, where the goal was to present more descriptive error and status messages about queries. This issue relied a lot on the backend side, and when planning the Sprint and creating the Sprint Backlog, we were informed that the needed information was already available. As mentioned in Subsection 4.2.2, it became

clear that the backend currently does not send the required information, but as another group is working on this, we agreed to keep the issue in the Product Backlog. This way, it can be assigned to a future Sprint, when the backend is ready.

The other unsolvable issue was issue #8 about displaying an error message related to JAVA HOME when trying to launch ECDAR. It was, however, not possible for us to recreate the error, so the issue was initially marked as "stuck". After discussions with the PO, the issue was discarded.

4.4 Sprint Retrospective

In this Sprint, we were overall good at documenting code changes as well as internal Scrum Meetings and the Scrum Meetings in the SoS group. When creating the Product Backlog, we used some time to thoroughly describe the issues, including a description of the bug or feature as well as its DoD. Halfway through the Sprint, we spent some time discussing how the structure of Scrum would fit into our report, and this was extremely helpful and made it easier for us to always be up to date with the different sections in our report throughout the entire Sprint. This contributed to maintaining a good overview of the process.

In the end of the Sprint Retrospective, we decided to formalize a couple of issues about things we should improve in the next Sprint, and this was then added to the Sprint Backlog of the second Sprint.

The things that we would like to improve were first of all to spend more time when planning out the Sprint. In the Sprint Planning for Sprint 1, we had not fully grasped Scrum as a method. To improve this, we started out by involving the PO in the Sprint Planning for the next Sprint. The PO participated in the Sprint Review, so at this meeting, the PO helped with estimating which issues could be part of the following Sprint, basically launching the Sprint Planning gradually and preparing us for the Sprint Planning in the SoS group.

Secondly, we worked in pairs to solve the different issues, which is generally a good thing that we would continue doing when possible. However, we ended up working in the same pairs and on the same issues and tasks all the time, meaning that some were very focused on the report, whereas others only looked at issues regarding the code. To preserve personal engagement in all aspects of the project, we therefore wanted to try to work in different pairs in the next Sprint, and we would consider shifting pairs halfway through the Sprint.

At the very end of the Sprint, we created two pull requests for the two issues described in Section 4.2, but no one had time to review these before the Sprint Review. This should definitely be changed in the next Sprint, as the solutions presented at the Sprint Review should be reviewed by the other groups first, and this was not the case for the first Sprint. We would also try to set aside time for reviewing pull requests from the other groups, so that all pull requests are reviewed and ready to be merged before the Sprint Review.

5 | Sprint 2: Onboarding II

Duration: 3/10-2022 - 14/10-2022

5.1 Sprint Planning

After Sprint 1, we had gained more knowledge of the Scrum framework and the codebase for ECDAR, and there was a clear structure of the overall collaboration between the different groups. This meant that the second Sprint focused more on solving issues and adding new functionalities.

Prior to the Sprint Planning in the SoS's group, we looked at the Product Backlog. The issues where we did not need further information from the backend were marked as "Ready". From these, the issues that the PO had prioritized as "Medium" or "High" were presented at the SoS group's Sprint Planning as a starting point for the Sprint Backlog. In the internal Sprint Planning, the Sprint Backlog was reviewed and we agreed on which issues should be a part of this Sprint. Our goal for this Sprint was to fix smaller bugs, but also to start working towards the implementation of simulation. Furthermore, some issues were added to the Sprint Backlog at the Retrospective for the first Sprint (Section 4.4). These were to improve how we as a group worked together and they were also a focus of this Sprint.

At the end of the Sprint Planning, we estimated the size of the issues for this Sprint with the use of planning poker. In planning poker, each group member has a set of cards with numbers. One issue is discussed, the group members will choose a card that represents an estimate of the issue size, and at the same time, all will turn their cards. Any outliers will be discussed, and the game will be repeated until there are no more outliers. For our planning poker, we used the modified Fibonacci sequence as explained in [13]. We used these estimates to decide how many group members should be assigned to the different issues, e.g., if an issue was estimated to be 8 or more, then several group members should

work in pairs. Most of the issues can be solved within the group, but for the simulation issue, we will have to work closely together with one of the Reveaal groups.

5.2 Implementation

In this Sprint, we solved several issues related to the GUI. We also prepared sketches of how we imagined the simulation to be illustrated. Our work is described in detail in this section.

5.2.1 Issue #25 – Auto-Hide Generated Components Tab

Issue description: The "Generated Components" tab on the left is always visible in the GUI, as shown in Figure 5.1. It is instead desired that when no components have been generated in the tab, then the "Generated Components" tab is not visible.

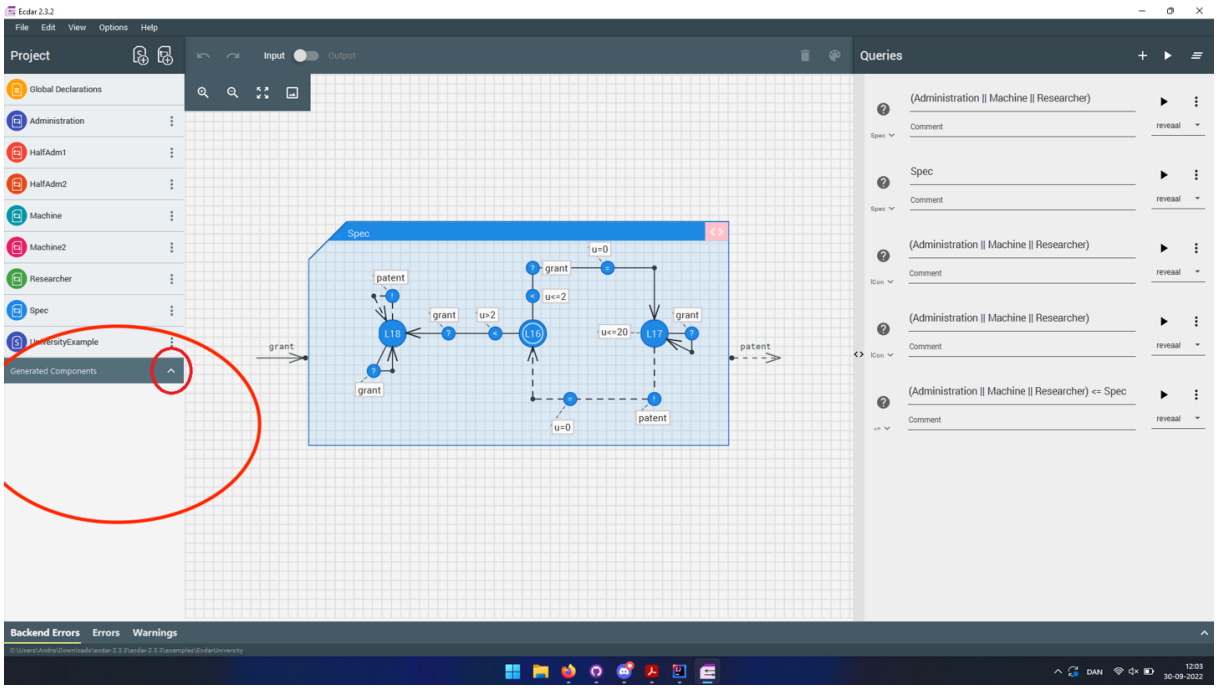


Figure 5.1. "Generated Components" tab displayed when no generated components exist.

Additionally, the arrow that indicates whether or not the generated components list should be shown is slightly deceptive. As indicated with the smaller red circle in Figure 5.1, the arrow points up, which could indicate that the generated components are the ones shown above, however, this is incorrect. Instead, the arrow should be displayed in a more accurate way.

Definition of Done: When no generated components have been created, the "Generated Components" tab should not be visible. Furthermore, the arrow should point to the left when no generated components are shown, and down when they are shown. **Implementation description:** This issue was solved by adding a conditional statement in the function that initializes the project pane. The statement checks whether or not the list with temporary components is empty. If the list is empty the visibility of the generated components tab is set to false, and if the list is not empty then the visibility is set to true. Furthermore, the icon code of the symbol on the tab has been changed, such that the arrow points to the left when the list is hidden and down when the list is displayed. This has yet to be tested in a usability test, as we can only assume how the user would understand this functionality.

5.2.2 Issue #21 – Align Width of Query Status Box

Issue description: As shown in Figure 5.2, the width of the box containing the status icon and the query type selector scales to fit the content inside. Because some query type names have a different number of characters and characters of different sizes, the width changes depending on which type is selected, causing an inconsistent layout.

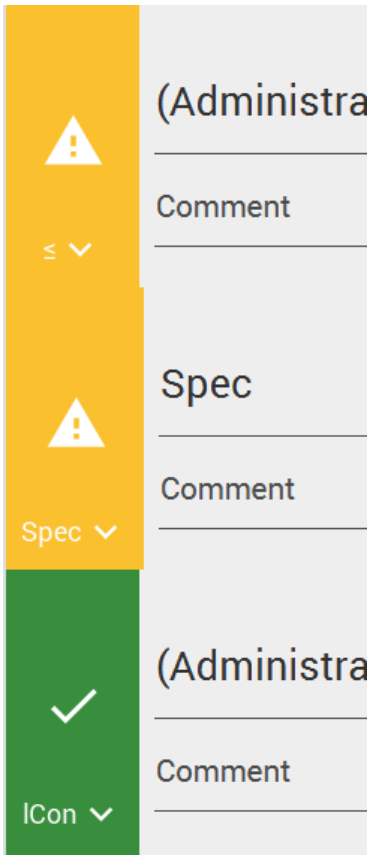


Figure 5.2. Inconsistent width of the box containing status icon and query type selector.

Definition of Done: Set the width to a fixed size that fits all query types.

Implementation description: Currently, the box has a minimum width of 2em and a maximum width of 4em. Simply setting the minimum width to 4em as well fixed the issue, 4em was chosen as it is the largest possible query. This, however, caused the icon and selector to be off-center so it was necessary to add an `alignment="CENTER"` attribute to the parent element.

5.2.3 Issue #19 – Prevent Query Running without Query Type

Issue description: When creating a new query, no query type is selected, but it is still possible to run the query by pressing enter, which results in a bug that causes what looks like an infinite loading state, as shown in Figure 5.3.

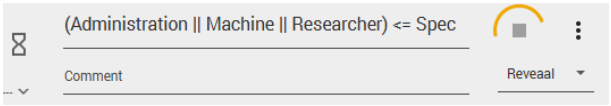


Figure 5.3. If no query type is selected and enter is pressed the GUI will attempt to run the query and end up in a seemingly infinite loading state.

Definition of Done: It should not be possible to run a query by pressing enter when no query type is chosen.

Implementation description: Currently, there is an if-statement that only checks if the enter key is pressed and if so, the `runQuery()` method is called. Simply by adding another condition to the if-statement that also checks if the query type is not equal to NULL ensures us that the `runQuery()` method cannot be called if no query type is chosen.

5.2.4 Sketches of Simulation

In order to prepare for simulation, we decided to design some ideas of how the simulation view could look, which we could then discuss with the PO.

Our initial idea of how a simulation could look is seen in Figure 5.4. We had the idea of showing possible transitions with a highlighting color on each component. Furthermore, we thought of implementing a button for running a simulation, however, this button was later removed, as we implemented the functionality to click the wanted transitions instead. In the sketch, we are in the upper middle location. We can follow the edge marked by the brown oval, and we will then end up in the dark blue location.

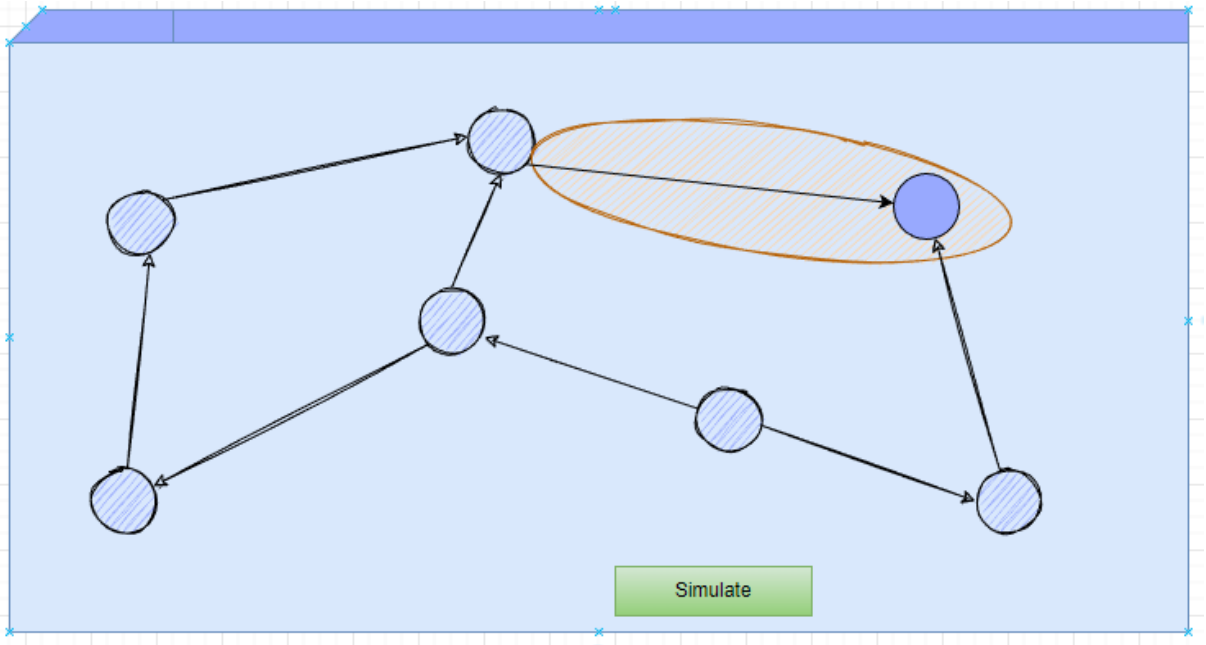


Figure 5.4. A simulation sketch that shows a possible transition.

The sketch seen in Figure 5.5 is an idea of how simulation could be used for reachability checks, to check whether a location is reachable from a given state, or the start state. In the sketch, the user is in the location in the left corner and we want to see if we can reach the location in the right corner. Since the location can be reached the path is highlighted.

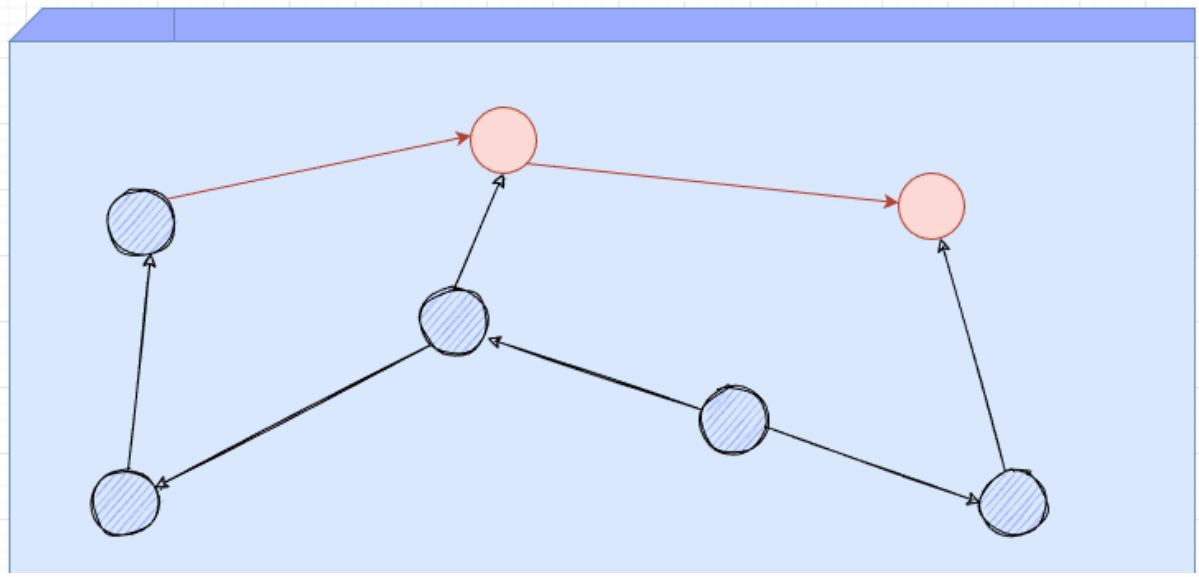


Figure 5.5. A simulation sketch that shows the highlighted path from a reachability check.

5.3 Sprint Review

At the Sprint Review for this Sprint, the PO was not present. For this Sprint the focus had been to fix different bugs to improve the GUI, and we also wanted to briefly begin on the simulation part.

For improving the GUI, we had three issues assigned to the Sprint Backlog, and these were all completed during the Sprint, as described in Section 5.2.

During the Sprint a lot of time was also spent on writing and correcting the different collaboration sections for the report together with the other ECDAR groups. This was supposed to be done at the end of this Sprint, and all groups therefore had to work towards this. However, this was not achieved, and at the SoS meeting it was decided to put these sections on hold.

For the simulation part, we had a short meeting with the PO to clarify what the purpose of the simulation was, and we then started to brainstorm how the simulation could be visualized in the GUI. At the end of the Sprint, we had a meeting with the backend group that works on simulation. Here we agreed on how the information should be passed between the frontend and backend in order to be ready for further development during the next Sprint.

5.4 Sprint Retrospective

During the Sprint Retrospective for the last Sprint we added some issues to the Sprint Backlog. Some of these were that we wanted to review pull requests earlier and to make sure that the same people did not work together with the exact same people as in the last Sprint. This was achieved during this Sprint as we had closed all pull requests before the Sprint Review and Retrospective. Also, in this Sprint we worked less in pairs and more individually. We also tried to keep up with the report as different issues were solved and we made sure that it was not the same people who wrote the same sections in the report as in the last Sprint.

The reason that we worked more individually was that not all group members were at the university as some worked from home some of the days. Even though we still held the Daily Scrums, it is more motivating and easier to communicate when we all are at the university together. We will therefore try to be there physically as much as possible.

For the next Sprint we would like to spend more time on testing, as this is a part of our

general DoD, as seen in Section 2.3. However, we have not spent as much time on testing as we should, and this should definitely be improved from now on. It can be difficult to test the user interface, when only small adjustments have been made. So to gain more knowledge on this, we will set up a meeting with the PO to get an introduction to the tests for the user interface that he has already created. Apart from this, we should spend more time on writing unit tests in general, as we want to ensure the stability of our solutions.

At the Sprint Retrospective in the SoS group, the communication between all of the groups was discussed. The groups usually communicate using Discord but all the groups tend to not answer when someone texts and this is problematic. This was therefore discussed and from now on we should all be better at answering on Discord to improve the communication. When a message has been seen it should be indicated by using an emoji to react to the message. Furthermore, pull requests, that are being reviewed, are to be marked by an emoji by the reviewer.

6 | Sprint 3: Simulation Preliminaries I

Duration: 17/10-2022 - 28/10-2022

6.1 Sprint Planning

For this Sprint the focus would for the most part be on simulation. Since we had a meeting with the backend group during the previous Sprint, we were now able to start working on the simulation part. In addition, we also had a special focus on testing when writing code, as we had not spent enough time on that up until this Sprint. This also meant that we had to go back and write test cases for earlier implemented solutions.

When using ECDAR, we sometimes notice small errors in the GUI, and these have continuously been added to the Product Backlog. This means that in most Sprints there will generally be some smaller issues that will improve the GUI, and this is also the case in this Sprint. The small issues, together with issues derived from the Retrospective of Sprint 2 and issues related to simulation, have been assigned to the Sprint Backlog for this Sprint, which can be seen in Figure 6.1. Here, the issues marked as "Draft" are the ones derived from the Retrospective.

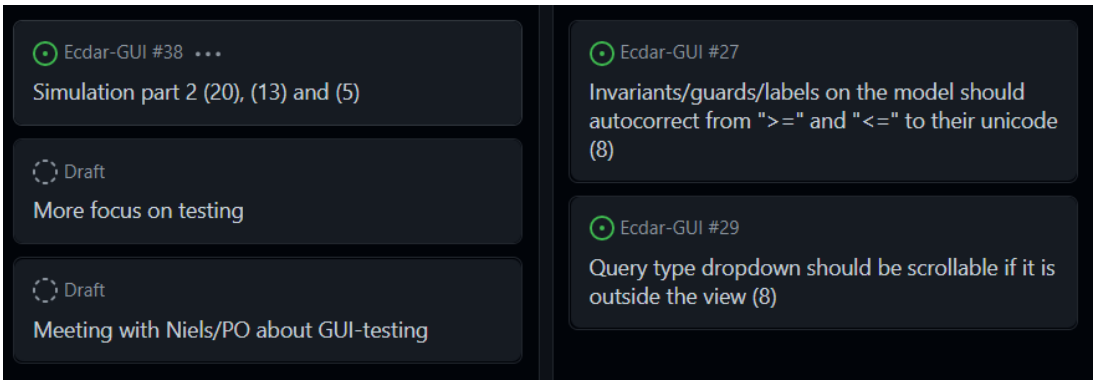


Figure 6.1. The issues assigned to the Sprint Backlog for the third Sprint.

To estimate the sizes of the different tasks in the Sprint Backlog, we again used planning poker. We discussed one issue at a time and when having agreed on an estimate, we wrote it next to the issue titles. The estimates can be seen in the parenthesis in the Sprint Backlog, Figure 6.1, these are an average of the groups estimates. The simulation issue was broken down into three smaller issues that covered the following: creating test data, sending requests to the backend, and being able to print the backend response in the GUI. It was necessary to create smaller issues to make a more precise and clear description of what should be done in this particular Sprint.

6.2 Implementation

This section describes how the issues mentioned above were solved. Not all issues were solved fully – but several Increments were made.

6.2.1 Issue #27 – Autocorrect Comparison Symbols II

Issue description: Similarly to Issue #12 (see Subsection 4.2.1), the invariants and guards on the models in the GUI should also autocorrect from ">=" and "<=" to "≥" and "≤".

Definition of Done: This issue is solved when the ECDAR GUI autocorrects ">=" and "<=" as the user types in these comparison operators in the invariant and guard fields on a model. When loading in already existing models the operators should also be displayed as "≥" and "≤".

Implementation description: This issue was not completely solved. The GUI is only able to autocorrect the symbols written in the nickname and invariant field, but not those written in the guard field. The text field was easy to find for the invariant and nickname field, whereas the guard text field could not be found, therefore we could not make any changes in that regard, at least during this Sprint.

To autocorrect the symbols two new methods were made in the `TagPresentation` class: `replaceSigns()` and `initializeTextAid2()`. The `replaceSigns()` method calls the `initializeTextAid2()` as shown in Snippet 6.1, selects the JFXTextField tag with the id textField using the lookup method, and sends it as a parameter to the `initializeTextAid2()` method. The `initializeTextAid2()` method then checks if the field's text contains ">=" or "<="; if so it will then be replaced into "≥" and "≤".

```
1 public void replaceSigns() {
2         initializeTextAid2((JFXTextField) lookup("#textField"));
3     }
```

Snippet 6.1. Method for replacing signs.

The `replaceSigns()` method is called in the `LocationPresentation` class when initializing the tags. For the query to be able to execute properly when it is sent to the backend driver the "≥" and "≤" signs must be in the format ">=" or "<=". Because of that, some changes were made in the `getInvariant()` method under the `Location` class, simply by converting the signs back to ">=" and "<=" as shown in Snippet 6.2.

```
1 public String getInvariant() {
2         return invariant.get().replace("\u2264", "<=").replace("\u2665", ">=");
3     }
```

Snippet 6.2. Converting the symbols back to ">=" or "<=".

6.2.2 Issue #29 – Fix Query Selection Menu Placement

Issue description: Currently when there are more queries than the size of the query panel can fit, parts of the query type drop-down for the query will disappear outside the application window. The user is therefore not able to choose these query types.

Definition of Done: When parts of the drop-down are not visible to the user, the drop-down should be displayed so that it is possible for the user to choose between all the different query types.

Implementation description: The issue was related to the scene and stage system in JavaFX, which is described below in relation to Figure 6.2. The system works such that when a drop-down is generated the specific size of the drop-down is unknown at the time of rendering. The space available is calculated with the size of the parent stage, as seen below in Snippet 6.3 in line 4. The solution was to estimate the size of the scene relative to the stage while rendering the properties for the drop-down, before showing it to the user as seen below from line 1-5. This meant that an if-statement could decide whether the drop-down would fit within the application screen, and in the case it could not, it will render it from the bottom of the app window to ensure that the drop-down would not surpass the limits of the application.

```
1 double windowHeight = this.getScene().getHeight();
2 Point2D Origin = this.localToScene(this.getWidth(), this.getHeight());
3 queryTypeDropDown.show(this);
4 if(Origin.getY()+queryTypeDropDown.getHeight() >= windowHeight){
5     queryTypeDropDown.show(JFXPopup.PopupVPosition.BOTTOM,
          ↪ JFXPopup.PopupHPosition.RIGHT, -55,-(Origin.getY()-windowHeight)-50);
6 }
7 else{
8     queryTypeDropDown.show(JFXPopup.PopupVPosition.TOP,
          ↪ JFXPopup.PopupHPosition.RIGHT, 16, 16);
9 }
```

Snippet 6.3. Solution to issue 29.

Stage and Scene in JavaFX

The stage is the main container in which scenes with nodes of content are stored, as shown in Figure 6.2. JavaFX has the main stage which has the width and height of the application window, while scenes are smaller containers, in which nodes can be placed depending on the scene size. This is what we used to calculate whether a drop-down node would exceed the stage window, by subtracting its placement in its scene from the size of the window, to calculate the remaining space within the main stage.

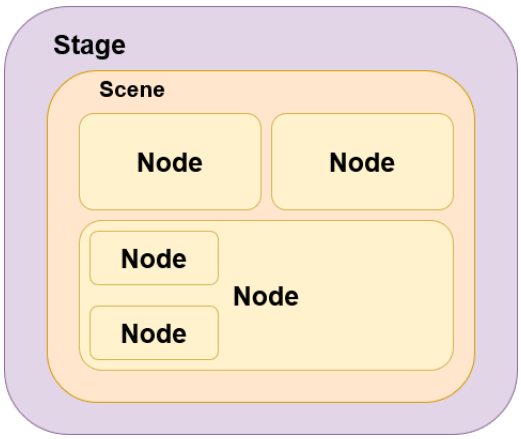


Figure 6.2. Stage, Scene, and Nodes in JavaFX.

6.2.3 Issue #38 – Send Simulation Start Request

Issue description: We have determined how the data for the simulation should be passed between the frontend and backend in the previous Sprint and can now start implementing it. The issue was divided into three smaller issues: creating test data, sending requests to the backend, and printing the response in the GUI.

Definition of Done: A `SimulationStartRequest` should be sent to the backend when the user switches from the editor to the simulator view and the response should be shown as text in the GUI.

Implementation description: Part of the preparation for the simulation was to update the implementation of gRPC to support the newly defined Protobuf schema. The new scheme defines two additional endpoints on the backend: `SimulationStartRequest` and `SimulationStepRequest`. Both endpoints return a `SimulationStepResponse` which contains information about the current locations, clock values, and available transitions.

However, the change to the Protobuf schema caused some existing functionality to break, so the first task was to fix this so the program was able to compile. The main issue was that the new schema now allows the backend to send detailed error messages to address issue #16 as described in Subsection 4.2.2, but the GUI does not support this feature yet. Fixing this involved changing the way a query is being sent to the backend and adding new error handling functionality for the response. It is only a temporary solution and it does not handle all types of errors perfectly but it is enough to run and use the program without problems. The issue was closed by another group.

The `BackendConnection` class has a field of type `EcdarBackendStub`, which is an auto-generated class that acts as a gRPC client that implements the services defined in the Protobuf services file. The method `EcdarBackendStub.startSimulation()`, which sends a request to the backend, takes two parameters: a `SimulationStartRequest` and a `StreamObserver`. The `SimulationStartRequest` contains all the information about the system such as the components, its edges, clock variables, etc., while the `StreamObserver` determines what to do when a response is returned.

6.3 Sprint Review

In the Sprint Review, we went through the issues that we had been working on in the Sprint. Only the developers were present at the meeting. In this Sprint, we had two visual issues, these being issues #27 and #29, which are about the presentation of the symbols $\leq$ and $\geq$ and about the placement of the query type selection drop-down, respectively. Finally, we had been working on issue #38 about simulation. In this section, we will describe for each issue, how the progress went.

Regarding issue #27, we managed to implement the conversion between the symbols written in one or two symbols functionally for invariants and nicknames. However, the implementation for guards will be a task for another Sprint, along with refactoring, testing,

and code reviews of the developed methods.

Regarding issue #29, the issue was solved to the degree where a solution was found. The PO also agreed that from a functional perspective, the issue appeared to be solved. Unfortunately, we did not complete the testing and reviewing of the code, so these will be tasks for another Sprint. In this Sprint, the PO agreed to give us an introduction to how the GUI testing that can simulate clicks and keyboard inputs works, so we will try to use that for testing in the future Sprints.

Finally, we worked on issue #38 regarding simulation. The simulation-related tasks for this Sprint were to create test data in collaboration with the relevant backend group, to send two different requests to the backend, and to present the response in a simple manner in the GUI. Unfortunately, the backend group seemed less interested in helping with test data, so this part was skipped. We did, however, successfully send requests to the backend to *start* the simulation, utilizing the newest version of the ProtoBuf protocol. We did not yet display the response in the GUI, but it could be seen in the terminal. This left sending *step* requests successfully to the backend for another Sprint. We also had to perform proper testing and code review on the Increments from this Sprint later.

From the SoS Sprint Review, several GUI-related tasks were derived. First of all, the group working with clock reduction requested a log that should be visible to the user in the GUI, along with a way to toggle whether clock reduction was enabled or not. The group working with cause of failure also requested a more visual presentation of errors, and they suggested that they could implement it themselves in the frontend, if we were unable to find the time for those less central tasks. These were not meant as requests, which we should act upon right away, but we will consider these for future Sprints.

6.4 Sprint Retrospective

In the Retrospective of the last Sprint (see Section 5.4), the communication between the different groups was discussed and we all agreed that it could be better. In this Sprint the communication had been improved as we were all better at answering each other on Discord, e.g., when pull requests had to be reviewed, where we now give a “thumbs up” to the message, if we start reviewing the pull request.

During the Sprint, we briefly started working on simulation and this also required communication with the other groups. The communication with the simulation backend group went well as we continuously made sure that we had the information from each other that we needed.

Overall, this Sprint was characterized by having quite a few hours available for working on the project as the courses took up most of the time. We, therefore, ended up creating a pull request regarding Issue #29 just before having the Sprint Review and Retrospective, so there was no time to finish the review of it before ending this Sprint.

Apart from this, we did not manage to write the implementation section in the report during the Sprint. However, to make sure that the missing sections were written before starting the next Sprint, we agreed on who would write which sections.

Overall, we had less time to work on the project than we had expected to have, which resulted in us not being able to finish everything in time. For the next Sprint, we should carefully consider how much time we have available when planning the Sprint. However, it is not always easy to predict during the Sprint Planning exactly how much time we can set aside for the project since our schedule can always change after the Sprint Planning, which would either give us more or less time to work on the project.

At the SoS Retrospective, we discussed that we should be better to delegate who would review the different pull requests as it is usually the same people who tend to review these. For now, the solution to this is that each individual should take a greater responsibility.

7 | Sprint 4: Simulation Preliminaries II

Duration: 31/10-2022 - 11/11-2022

7.1 Sprint Planning

Not all issues from the Sprint Backlog for Sprint 3 were fully implemented, so for this Sprint the focus was on these issues along with some more simulation related issues. More specifically, we worked on issue #27 and #38, as described in Section 6.2, along with issue #50 regarding the simulation view in the GUI. Altogether, these issues supported the overall goal of being able to do simulation in the ECDAR GUI.

To perform this, we split into three pairs, each working on one of the issues. As we had many courses in the time frame of this Sprint, we expected to be able to finish issue #27 at least, and to prepare for the simulations requests described in issue #38. We knew from the SoS' meeting that other developers on the ECDAR project had been working on refactoring the backend driver, so before we could start sending requests to the backend, these changes should be incorporated in a functional manner. We expected this to be quite a voluminous task. Furthermore, we expected to be able to improve the simulation view as described in issue #50, all contributing to the implementation of simulation.

From the SoS' meeting we also knew that both the reachability and simulation groups were still working on the responses, they should send, so we could still not get a proper response from the backend for testing our implementations. In the case that this potential problem became a block for our feature development, we would implement a stub, which would be a new temporary class that would handle sending and receiving demo requests and responses. This way we could ensure the continued development.

7.2 Implementation

This section describes how the three issues (#27, #38, and #50) were solved. It describes how issue #27 was solved and tested, and how the refactoring was implemented in our fork in issue #38. Regarding issue #50, the complexity of the solution is once again revealed, as the issue could not be solved fully within the Sprint.

7.2.1 Issue #27 – Autocorrect Comparison Symbols II

Issue description: Similarly to Issue #12, which is described in Subsection 4.2.1, the invariants and guards on the models in the GUI should also autocorrect from ">=" and "<=" to "≥" and "≤".

Definition of Done: This issue is solved when the ECDAR GUI autocorrects ">=" and "<=" as the user types in these comparison operators in the invariant and guard fields on a model. When already existing models the operators should also be displayed as "≥" and "≤".

Implementation description: As the issue was not completely solved in Sprint 3, we continued working on it in this Sprint. The remaining parts were solving the issue for guards, refactoring the code, and testing.

The reason why it was more complicated to add the event listener to the guard's label, was that the guard was "disguised" as a nail, and hence the changes should be in the `NailPresentation` class. As the guard here was also presented in the shape of a `tag`, the `replaceSigns()` method made for invariants and nicknames (see Subsection 6.2.1) could simply be called on the `propertyTag` in the `NailPresentation` class.

When the content of the guard field should be sent to the backend driver, it should be in this format ">=" and "<=", like for the content of the invariant field. Therefore, the `getGuard()` method placed in the `DisplayableEdge` class has been modified slightly, so it converts the signs back to ">=" and "<=" before adding the value of the guard to requests for the engines.

As the methods to perform the conversion between the symbols and Unicode characters are used across different classes, we decided to make a refactorization of the code, and gather the methods in a helper class called `StringHelper`. This class contains the conversion logic that was described in Subsection 4.2.1, now represented by `static` methods.

We tested the methods in the `StringHelper` class by making two parameterized tests,

one of which is shown in Snippet 7.1. The input values and expected output values are formatted as semi-colon separated strings (line 3-7), as the `ValueSource`-annotated element in line 2 can only contain an array of simple types, where one will be used for each test. This meant that we had to combine the different strings for each test into one, which was then split when the test was launched (line 10-12). Alternatively, we could have used the `Parameterized.Parameters` attribute, which allows for making more complex input types. However, this strategy requires a test runner in order to execute the tests, and we therefore decided to keep it simple and use the `ValueSource` attribute. We made different tests both with and without the symbols to test both conversion algorithms. The test from symbols to Unicode characters is shown in line 9-15. A similar test was made for converting Unicode characters back to separate symbols again.

```
1 @ParameterizedTest
2 @ValueSource(strings = {
3         "2 >= 4;2 \u2265 4",
4         "2 <= 4;2 \u2264 4",
5         "2 == 4;2 == 4",
6         "2 > 4;2 > 4",
7         "2 < 4;2 < 4",
8 })
9 void convertSymbolsToUnicode(String inputAndExpectedOutput) {
10     var split = inputAndExpectedOutput.split(";");
11     var input = split[0];
12     var expectedOutput = split[1];
13     var actualOutput = StringHelper.ConvertSymbolsToUnicode(input);
14     assertEquals(expectedOutput, actualOutput);
15 }
```

Snippet 7.1. Test of string conversions.

All tests passed, so we concluded that the code works as intended.

7.2.2 Issue #38 – Refactoring of the BackendDriver

Issue description: Issue #38 concerns the backend communication of simulation and was initiated last Sprint. However, as discussed with the PO, a refactoring of the `BackendDriver` class was needed, since it contained a lot of functionality that could be restructured into separate classes. It is not desired to have all the functionality in one class, because it makes the code for this class less cohesive, which means that it is hard to discover what code is related to what functionality, and overall makes it hard to keep track of the code.

Definition of Done: The `BackendDriver` class should be refactored into separate classes, such that each class is only responsible for exactly one part of the backend communication. This means that the refactoring should introduce new classes that are responsible for opening and closing backend connections, handling gRPC requests, handling query requests, and handling simulation requests.

Implementation description: The refactoring was implemented by decoupling the code from the `BackendDriver` class and thereby introducing four new classes:

- *GrpcRequest*: A more generic class for gRPC requests, that can now be used for both query and simulation requests.
- *QueryHandler*: This is the class that is responsible for sending the query that is being executed to a backend engine and, afterward, handling the response from the backend.
- *SimulationHandler*: This class is responsible for most backend communication related to simulation. This includes sending a start request to the backend when simulation starts, as well as sending step requests and reachability requests to the backend. As of now, most of these features had not been implemented yet, but they were implemented in Sprint 5 (see Subsection 8.2.2 and Subsection 8.2.4).
- *BackendConnection*: Handles the opening and closing of backend connections together with the BackendDriver class.

7.2.3 Issue #50 – Visible Part of Simulation

Issue description: A simulation window in the GUI must be implemented, in order for the user to be able to interact with the different components during simulation. This issue has been divided into two smaller parts, being:

1. Implement a simple layout that only shows the components that are relevant to the current simulation. This should be done, as currently when entering the simulation view with a query regarding a subset of the components, all components are shown in the simulation window.
2. Add simple functionality to highlight edges from the initial location in each component, as well as the initial location itself. This is to prepare for the visualization of the current state and the possible transitions that can be made. This information should be received from the engines.

Definition of Done:
In accordance with the issue description, the DoD for this issue is also divided into two:

1. Only the relevant components, meaning the components involved in the query, are presented to the user, in a simple way.
2. In each component in the simulation window the initial location is highlighted together with the edges leaving the initial state.

Implementation description:
In this Sprint, we were not able to implement the second part of this issue, as it turned out to be a bigger task than expected. This part was implemented in Subsection 9.2.7.

The first sub-issue has been implemented as follows. When the user enters the simulation view they have to choose one of their queries and press "Start Simulation". Whenever this button is clicked, a method for retrieving the chosen query will be invoked in the `SimulationInitializationDialogController` class (see Snippet 7.2) by the "onMousePressed" event on the "Start Simulation" button. On line 2 in `GetListOfComponentsToSimulate()` the chosen query from `simulationComboBox` is retrieved and stored in the string called `componentsToSimulate`.

Line 3-4 will look through the `componentsToSimulate` and match the substrings that satisfy the regex: "([\\w]*)". This regex will match words that contain letters, numbers or "_" and in the while-loop from line 7 to 11 these words or substrings will be added to the local list called `listOfComponentsToSimulate`. On line 12 this list is then assigned to a static list called `ListOfComponents`. We were unsure how to get access to the list from the `SimulatorController` class. Therefore, we started out by making the property static in this issue.

In issue 107 (see Subsection 9.2.9), the property was moved to another class, that both the `SimulationInitializationDialogController` and the `SimulatorController` classes can access.

```
1 public void GetListOfComponentsToSimulate(){
2     String componentsToSimulate =
          ↪ simulationComboBox.getSelectionModel().getSelectedItem();
3     Pattern pattern = Pattern.compile("([\\w]*)", Pattern.CASE_INSENSITIVE);
4     Matcher matcher = pattern.matcher(componentsToSimulate);
5     List<String> listOfComponentsToSimulate = new ArrayList<>();
6
7     while(matcher.find()){
8         if(matcher.group().length() != 0){
9             listOfComponentsToSimulate.add(matcher.group());
10        }
11    }
12    ListOfComponents = listOfComponentsToSimulate;
13 }
```

Snippet 7.2. `GetListOfComponentsToSimulate()` **for retrieving the involved components in a chosen query.**

Another method in a different class then iterates through all the existing components and only chooses the components where the name of the component is in the static `ListOfComponents` list. A copy of these components will be made to ensure that the components in the editor view will not be affected by the simulation as a copy by reference would mean that the GUI changes applied in the simulation would occur in the editor view, as we would be referencing the same objects. These will then be returned and presented in the simulation view.

7.3 Sprint Review

In this Sprint, we worked on issues that we had not finished in Sprint 3. To recap, these were issues #27 and #38 regarding autocorrection of specific mathematical symbols and refactoring of the backend driver, respectively. We also worked on issue #50 regarding the visual foundation for the implementation of simulation.

Starting with issue #27, we successfully identified how the text in guards was accessed, and thus we were able to implement the final functionality. We also set up parameterized tests, and the code was reviewed by members of other groups. It could therefore be merged into the main branch, and the issue was closed.

Regarding issue #38, the backend driver was refactored, but we did not have time to test and review the code. This had to be done in the next Sprint, as it was blocking solutions to some other issues from being merged into the main branch, as they depended on the

changes made to solve this issue. Finally, we still needed to set up the simulation requests, which was the true purpose of the issue.

For the final issue, issue #50, we completed the first part about implementing a simple layout for the simulation that only shows the relevant components. We did not manage to add functionality to highlight transitions and states.

In the joint Sprint Review, it became clear that the Cause of Failure-group, which had been implementing error messages in the GUI, had identified different improvements that could be made to the GUI. As they were running out of issues to solve themselves, we agreed that they should add their GUI-related issues to the GUI Product Backlog, and that we could then coordinate our efforts, as they would like to work on the GUI in the next Sprint.

We also learned from this meeting that the group which had been working on multithreading in the Reveaal engine had completed their task, and therefore it was ready to be integrated in the GUI. The implementation of multithreading in the GUI will be handled by the team responsible for its development in the coming sprint.

7.4 Sprint Retrospective

In this Sprint, we worked in pairs throughout the entire Sprint. This allowed us to work on all issues simultaneously. We decided to work in new pairs, which had a positive influence on the work and the *esprit de corps*.

We also had good communication with the Cause of Failure group, which was also working on GUI-issues. As they branched out from one of the branches, we were still working on, some coordination was needed in regards to how to merge their changes into the main branch. This went very well, facilitated by easy access to each other, as we were able to coordinate our work on both Discord and by stopping by their group room.

In this Sprint, we were very challenged by the amount of incomplete legacy code that we were faced with when attempting to implement simulation in the GUI. As most of the code for the simulation view was implemented by a student group, which was no longer connected to the project, we did not have the option of reaching out when the code did not make sense. Trying to sort out the code to remove dead code and to simplify the work took a lot of our time, blocking the development of new features. In the next Sprint we will focus heavily on implementing the functionality described in the issues we work on in this Sprint. Cleaning up legacy code was added to the Product Backlog and would

be an issue for later Sprints.

Another thing that we could improve in the future Sprints was breaking down issues into smaller parts. An example of an issue where we should have done this can be found in issue #50, which contained two very independent tasks, that had to be described completely separately. In the next Sprint we would start by splitting up this issue into smaller issues. This would increase our chances of completing the issues within the time frame they have been assigned to. It would also help us make better estimates through planning poker, as we suffered from bad estimates in this Sprint. In general, we underestimated the workload of the three issues, and we expected that by splitting them up into smaller issues, it would be easier to make a more accurate estimate.

8 | Sprint 5: Simulation and Reachability I

Duration: 14/11-2022 - 25/11-2022

8.1 Sprint Planning

At the SoS Sprint Planning, the reachability group told us that they were now able to give a yes/no response to reachability checks, given a start state and an end state. Therefore, the main goal for this Sprint was to implement the reachability check in the simulation view. This introduced the following two new issues to the Product Backlog. What exactly these issues consist of is further explained in Section 8.2.

- *Issue #76:* Reachability: show response (Y/N)
- *Issue #89:* Reachability: send request

Additionally, the GUI refactoring of the backend communication from the previous Sprint had still not been merged into our main branch, and this was starting to become a bottleneck for both us and other groups. Therefore, we had to highly prioritize getting the open pull requests merged. This required a clean up of several branches, as well as resolving conflicts and comments from the external reviewers.

A focus point for this Sprint was to split our issues into smaller tasks. As mentioned in the Scrum guide, Product Backlog items should be decomposed into smaller work items of one day or less. Doing this will ensure that each task is more manageable and that the Product Backlog items are turned into Increments of value [10]. Figure 8.1 shows the issues for this Sprint.

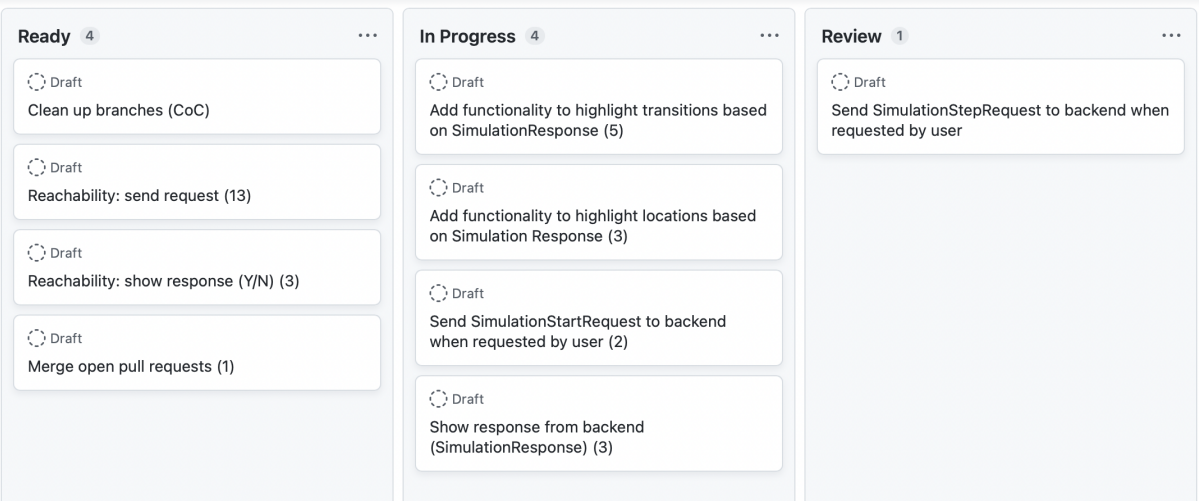


Figure 8.1. Issues for Sprint 5.

Figure 8.1 shows that we were still working on implementing simulation in the GUI. Here, we also split the tasks into smaller issues, and we could see that some work had already been done in regards to, e.g., sending the simulation start request. Therefore, several issues were already "In Progress" at the beginning of the Sprint.

8.2 Implementation

This section documents how the Increments were made. In the Sprint Planning, we were very concerned with splitting the issues into as small chunks as possible – this would however give some redundancy in the implementation descriptions of issues #84 and #85, so these issues have been described together. The issue descriptions are ordered such that all simulation related issues are described first, followed by issues regarding the reachability feature.

8.2.1 Issues #84 and #85 – Highlight Simulation Locations and Transitions

Issue description: When the simulation is running, the backend responds with a `SimulationStepResponse` object which contains information about the current locations, available transitions, and clock variables. The current locations and available transitions should be highlighted in the simulator view.

Definition of Done: The issues can be closed when the current locations and available transitions are highlighted in orange after receiving a `SimulationStepResponse`. The

color orange was first chosen as it is the color used in the editor view.

Implementation description: The implementation for highlighting the locations and transitions are very similar, so only the implementation of locations is described in the report. The method responsible for highlighting locations can be seen in Snippet 8.1. The method is located in the `SimulatorOverviewController` class, which controls the middle window of the simulator view. This class has a field, `processPresentations`, that contains all processes, meaning components, shown in the middle window. Each process has a `ProcessController` that supports the method `highlightLocation()` that takes the id of the location to highlight as a parameter. To highlight the locations of the current state, we iterate through its locations, as seen in line 4, and find the `processPresentations`, in which the location is placed (lines 5 to 6). Then in line 7, we call `highlightLocation()` method on the correct `processPresentation` with the id of the location as argument.

```
1 public void highlightProcessState(final SimulationState state) {
2     if (state == null) return;
3
4     for(var loc : state.getLocations()){
5         processPresentations.values().stream()
6             .filter(p ->
                  ↪ p.getController().getComponent().getName().equals(loc.getKey()))
7             .forEach(p -> p.getController().highlightLocation(loc.getValue()));
8     }
9 }
```

Snippet 8.1. `highlightProcessState()` **highlights the current locations of a given state.**

By adding a listener to SimulationHandler.currentState that calls highlightProcessState, the current locations are highlighted in orange every time a new response is received from the backend as shown in Figure 8.2.

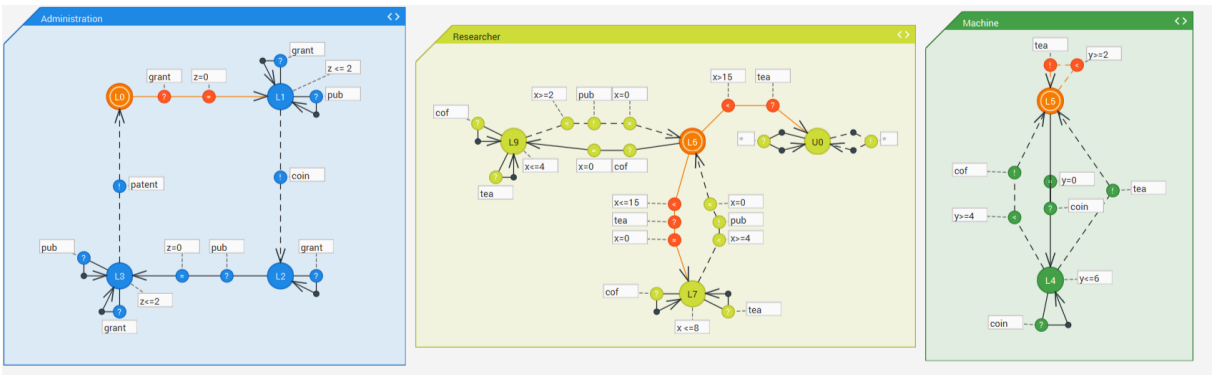


Figure 8.2. Highlighting of current locations and available transitions.

8.2.2 Issue #86 – Send Simulation Start Request II

Issue description: This issue is an extension of issue #38 and includes adding functionality to the "reset simulation" button, some refactoring of the controller responsible for starting the simulation, and swapping example data with real data from the backend.

Definition of done: The button shown in Figure 8.3 should reset the simulation and send a new `SimulationStartRequest`. The method `willShow()` in `SimulationController` should be refactored so it is easily readable. Additionally, the example data in the `SimulationHandler` class should be deleted and actual data received from the backend is to be used instead.

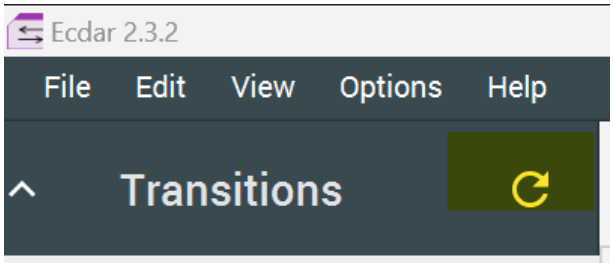


Figure 8.3. Highlight of the reset simulation button.

Implementation description: Since the button for resetting the simulation already existed, we only needed to create a method that calls `SimulationHandler.resetToInitialLocation()`. As seen in Snippet 8.2, a new method called `restartSimulation()` was created with the annotation `@FXML` that tags the method as accessible in the related FXML file.

```
1 @FXML
2 private void restartSimulation() {
3     simulationHandler.resetToInitialLocation();
4 }
```

Snippet 8.2. Snippet of the SimEdgePresentation class.

Now, we can call the method by adding an `onMouseClicked` attribute to the button in the FXML file.

Deleting example data was done by simply deleting a line of code that overwrites the actual response and deleting the method that created the example data.

8.2.3 Issue #87 – Show Simulation Trace Log

Issue description: To improve the practicality of the simulation, the user should be able to see and return to previous states of the simulation without having to restart the whole simulation.

Definition of done: It will be implemented as a list of states in chronological order, oldest at the top, in the left side pane of the simulation view. Each item of the list should show information about the locations and clock values. Clicking on a state will rewind the simulation to that state.

Implementation description: `SimulationHandler` has a property called `traceLog` which is an `ObservableList` of simulation states. Each response received from the backend, is added to this list. As a result of being an `ObservableList` it is possible to listen for changes so when a state is added to the list, the method `insertTraceState` is called with the newly added state as parameter. This method is responsible for creating a presentation for that state which contains the names of the locations. Additionally, an `EventHandler` is set so that when a user clicks on that state, it will be set as the current state and the proceeding states will be deleted. As shown in Figure 8.4, the trace log only contains the names of the locations and not the clock values. The rest of the work was completed in Sprint 6 and can be seen in Subsection 9.2.2.

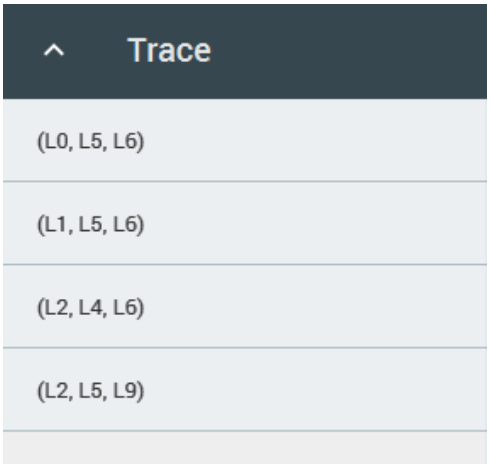


Figure 8.4. Trace log shown in left pane of the simulator.

8.2.4 Issue #88 – Send Simulation Step Request

Issue description: A `SimulationStepRequest` should be sent to the backend when a step is taken in the simulation.

Definition of done: The user should be able to walk through the simulation by clicking on available transitions in the visual presentation of the models. When clicked, the selected edge will be used to build a `SimulationStepRequest`, which will be sent to the backend.

Implementation description: To make an edge clickable, we can use the built-in method `setOnMouseClicked` that defines a function to be called when the user clicks on it. Since only some transitions are enabled, it is necessary to check if the clicked transition is actually enabled as shown at line 2 of Snippet 8.3. If the edge is enabled, the property `selectedEdge` is set and the method `nextStep` is called. This method builds the `SimulationStepRequest`, which contains information about the simulation and the chosen edge and sends it to the backend. Finally, to improve the user experience, we added a feature so that when you hover over an enabled transition, the cursor will change to a hand to indicate that it is actually clickable. The clickable area is the same size as the object, but as the objects are rather small, testing whether this is a problem could be done in the future.

```
1 this.setOnMouseClicked(event -> {
2     if (Ecdar.getSimulationHandler().currentState.get().getEdges()
        ↪ .contains(new Pair<>(component.getName(), edge.getId()))) {
3         Ecdar.getSimulationHandler().selectedEdge.set(edge);
4         Ecdar.getSimulationHandler().nextStep();
5     }
6 });
```

Snippet 8.3. Snippet of the SimEdgePresentation class.

8.2.5 Issue #89 – Send Reachability Request

Issue description: Before solving this issue, right-clicking a location in the editor view in the GUI would reveal a drop-down menu where the option to perform a reachability check was located. The logic was not implemented though, so nothing happened when it was clicked. Furthermore, it would be difficult to implement the logic as it would not be clear which components should be involved in the check. To counter this, it should instead be possible to right-click on the locations in the simulation view and select the reachability check option from a drop-down menu there. When the reachability check option is chosen, the simulation view should be able to send a reachability request starting from the initial or current state of the simulation to the backend engine.

Definition of Done: This issue is closed when the user is able to right-click on a location

in the simulation view and select the reachability check option that triggers a reachability request to be sent to the backend engine, starting from the initial or current state to the location clicked.

Implementation description: To solve this issue, we first split it into two parts, which different people could work on simultaneously:

1. The reachability check option should be moved from the drop-down menu in the editor view to a new drop-down menu in the simulation view.
2. The reachability request should be sent to the backend. The start state should be the initial state, which was the backend default.
3. The start state of the reachability request should be the current state of the simulation.

In order to show the drop-down menu when right-clicking on the locations in the simulation view, some inspiration was taken from the existing drop-down menu, which was created in the `LocationPresentation` and `LocationController` classes. To make the new drop-down for the simulation view, we had to make some changes in the corresponding simulation classes, i.e., the `SimLocationPresentation` and `SimLocationController` classes. The method `initializeDropDownMenu()`, as seen in Snippet 8.4, was added to the `SimLocationController`, where it generates the drop-down when right-clicking a location.

When the reachability check option is chosen, a request is sent to the backend to see if the location can be reached. This is done using the `getLocationReachableQuery()` method placed in the `BackendHelper` class, where it generates and returns a reachability query based on the given location and component.

```
1 public void initializeDropDownMenu(){
2         dropDownMenu = new DropDownMenu(root);
3 
4         dropDownMenu.addClickableListElement("Is " + getLocation().getId() + "
              ↪ reachable?", event -> {
5 
6             final String reachabilityQuery =
                  ↪ BackendHelper.getLocationReachableQuery(getLocation(),
                  ↪ getComponent());
7             final String reachabilityComment = "Is " +
                  ↪ getLocation().getMostDescriptiveIdentifier() + " reachable?";
8 
9             final Query query = new Query(reachabilityQuery, reachabilityComment,
                  ↪ QueryState.UNKNOWN);
10            query.setType(QueryType.REACHABILITY)
11            Ecdar.getQueryExecutor().executeQuery(query);
12            dropDownMenu.hide();
13        });
14    }
```

Snippet 8.4. The method `initializeDropDownMenu` generates a drop-down menu when right-clicking a location.

To ensure that we sent valid requests to the backend, we had to update the `getLocationReachableQuery()` method. The group working on executing reachability checks in the backend engine provided us with a grammar of how the requests should be formatted. First, there should be a reachability prefix followed by a colon and the component composition being investigated, which in our case corresponds to the query used to start the simulation. This should be followed by an arrow, "`->`" after which the start and end states could be specified with locations and clock constraints, separated by a semi-colon. The start state could be omitted, in which case the initial state of the system would be considered as the start state. Examples of valid queries could be the following:

```
reachability:  M1 || M2 -> [L1, L4](y<3);[L2, _]()
 reachability:  (M1 && M2) || M3 -> [_, L4, _]()
```

In both of these queries, some locations in the end states are represented by underscore symbols, indicating that they are what the backend group calls "partial" states. In practice, this means that we are not concerned with the value they obtain at all. If the brackets for clock constraints are empty, it correspondingly means that we have no clock constraints which the state must satisfy.

When we implemented this in the GUI, we used a top down approach to compose the reachability query string, as seen in Snippet 8.5. We added the start simulation query in line 4, but at this point, we only had access to the components, so we had to put the query together again. This was improved later when issue 98 was solved. In lines 4 and 5, we then added the arrow and the end state, where only the selected location was specified – the end locations in the other components were partial. Finally, in line 6 we appended empty brackets, as the GUI does not yet support adding clock constraints to reachability checks in the simulation view. This was further investigated in Sprint 6.

```
1 public static String getLocationReachableQuery(final Location endLocation, final
      ↪ Component component) {
2     var stringBuilder = new StringBuilder();
3     stringBuilder.append(getSimulationQueryString());
4     stringBuilder.append(" -> ");
5     stringBuilder.append(getEndStateString(component.getName(),
          ↪ endLocation.getId()));
6     stringBuilder.append("()");
7     return stringBuilder.toString();
8 }
```

Snippet 8.5. `getLocationReachableQuery` **returns a reachability query string for a given location.**

Hereby, the functionality to right-click a location in the simulation view and perform a reachability check from the initial state to this location had been implemented. Due to lacking knowledge on how to retrieve state information, the functionality to perform reachability checks from the current state was implemented later on.

Testing: Several tests were created to test the `getLocationReachableQuery()` method that is used to generate the query that is sent to the backend. Snippet 8.6 shows the test of whether the syntax of the reachability query string is correct. For that, the `assertTrue` method is used to check if the result matches the regular expression defined in line 3. The result variable, as seen in line 15, is the reachability query string generated by the `getLocationReachableQuery` method as described before.

```
1 @Test
2 void reachabilityQuerySyntaxTestSuccess() {
3     var regex = "([a-zA-Z]\\w*)([|][|][a-zA-Z]\\w*)*\\s+\\->\\s+\\[(\\w*)
          ↪ (,(\\w)*)*\\]\\([a-zA-Z0-9_<>=]*\\)(;\\[(\\w*)(,(\\w)*)\\]\\
          ↪ ([a-zA-Z0-9_<>=]*\\))*";
4
5     var location = new Location();
6     location.setId("L1");
7     var component = new Component();
```

```
8     component.setName("C1");
9
10    SimulationInitializationDialogController.ListOfComponents.clear();
11    SimulationInitializationDialogController.ListOfComponents.add("C1");
12    SimulationInitializationDialogController.ListOfComponents.add("C2");
13    SimulationInitializationDialogController.ListOfComponents.add("C3");
14
15    var result = BackendHelper.getLocationReachableQuery(location, component);
16    assertTrue(result.matches(regex));
17 }
```

Snippet 8.6. Testing the correctness of the reachability query string constructed by `getLocationReachableQuery`.

Besides checking the syntax of the reachability query string, other tests were made to ensure that the locations are in the correct positions according to the reachability query string. The test method shown in Snippet 8.7 checks if the location is placed in the correct position in the query, assuming that it was located in the component with the name C1. According to the order in which the components are added to the `ListOfComponents` at line 9-11 the location with the id L1 should be placed right after the '[' which is the `indexOfLocation` seen at line 14. Then the `assertEquals()` method is used to check whether the expected output equals the actual location id. Other tests similar to `reachabilityQueryLocationPosition1TestSuccess` are made where the components have been added in a different order to the `ListOfComponents`.

```
1 @Test
2  void reachabilityQueryLocationPosition1TestSuccess() {
3     var location = new Location();
4     location.setId("L1");
5     var component = new Component();
6     component.setName("C1");
7
8     SimulationInitializationDialogController.ListOfComponents.clear();
9     SimulationInitializationDialogController.ListOfComponents.add("C1");
10    SimulationInitializationDialogController.ListOfComponents.add("C2");
11    SimulationInitializationDialogController.ListOfComponents.add("C3");
12
13    var result = BackendHelper.getLocationReachableQuery(location, component);
14    var indexOfLocation = result.indexOf('[') + 1;
15    var output = result.charAt(indexOfLocation);
16    assertEquals(output, location.getId().charAt(0));
17 }
```

Snippet 8.7. Testing the position of the locations in the reachability query string.

The last test checks whether the number of locations is correct (see Snippet 8.8). In this specific scenario, there is only one location with the id L1 in the C1 component. When the `getLocationReachableQuery` method generates the reachability query string, it places an underscore on the place where there is no assigned location. The expected output is `C2||C1||C3||C4 -> [_,L1,_,_]()`, so the string is expected to contain three underscores in total, each representing arbitrary locations, plus the L1 location. That gives four represented locations in total. The `assertEquals()` method is used to check whether the size of the `ListOfComponents` equals the number of locations.

```
1  @Test
2      void reachabilityQueryNumberOfLocationsTestSuccess() {
3          var location = new Location();
4          location.setId("L1");
5          var component = new Component();
6          component.setName("C1");
7
8          SimulationInitializationDialogController.ListOfComponents.clear();
9          SimulationInitializationDialogController.ListOfComponents.add("C2");
10         SimulationInitializationDialogController.ListOfComponents.add("C1");
11         SimulationInitializationDialogController.ListOfComponents.add("C3");
12         SimulationInitializationDialogController.ListOfComponents.add("C4");
13
14         var query = BackendHelper.getLocationReachableQuery(location, component);
15         int underscoreCount = 0;
16         for (int i = 0; i < query.length(); i++) {
17             if (query.charAt(i) == '_') {
18                 underscoreCount++;
19             }
20         }
21
22         assertEquals(SimulationInitializationDialogController.ListOfComponents.size(),
               ↪ underscoreCount + 1);
23     }
```

Snippet 8.8. Testing that the number of locations in the reachability query string is correct.

8.2.6 Issue #76 – Show Reachability Response

Issue description: When a reachability response has been sent to the GUI, the user would need some sort of manifestation of the response of the request in the GUI. The backend responds with a message containing the result (reachable or not) and the

transitions taken in order to get to the requested location. For a start, the response should be displayed to the user in a pop-up that simply informed the user of whether the location was reachable. In Sprint 6, the paths were also highlighted on the components in the simulation view.

Definition of Done: The response from the backend is shown as a small pop-up message in the bottom of the simulation view. If the location can be reached, the message will say that this is the case and vice versa.

Implementation description: To show the message in a pop-up, the already existing static method called `showToast()` was used. This method takes a string as input and shows the message in the bottom of the GUI. `showToast()` is now called when the reachability response is received. If no response is received from the backend, the `showToast()` is called in a method that handles backend errors related to queries of type "reachability".

8.3 Sprint Review

The main goal for this Sprint was to make sure that the main branch was up to date and to clean up the many branches that we had that were not being used. It was therefore important to merge the open pull requests containing all the refactoring and to delete the branches that were outdated and unused. This was solved during the Sprint, which made it easier to continue working on new branches.

Another focus point for this Sprint was to start implementing reachability checks in the simulation view. This meant that a request should be sent to the backend to check if a location could be reached from the initial or current state (issue #89). The response should then be presented in the GUI (issue #76). The functionality that solves these two issues were partly implemented during the Sprint, and at the last day of the Sprint, a pull request for these were sent to the other groups. However, the issues were not merged to the main branch within this Sprint, as we still needed one external review before meeting the external DoD. The pull request must therefore be merged into the main branch in the next Sprint. The same was the case for all of the simulation related issues solved within this Sprint.

The only thing from the two aforementioned issues that was not implemented was the ability to perform reachability checks from the current state in the simulation view. When implementing the functionality for reachability checks, we discovered that we were not ready to implement the functionality for sending a request from a given location other than

the initial state. This was due to lacking knowledge of how to get the state information, and we had not considered this at the Sprint Planning. Another issue was therefore created for this.

For this Sprint we also wanted to implement the visualization of the simulation response (issue #87). This is still in progress as clock values still have not been considered in the simulation.

8.4 Sprint Retrospective

As mentioned in Section 8.1, a focus point for this Sprint was to decompose issues into smaller tasks. This was overall a great success, and it resulted in each task becoming more manageable. Decomposing the issues also made it easier to correctly determine the number of points to give each task for planning poker. Additionally, having better defined tasks in our backlog for this Sprint made it easier to assign tasks to group members and to reduce the amount of uncertainty that arises from each task.

Overall, this has been a productive Sprint, where a lot of progress has been made towards simulation and reachability. We discussed during the Retrospective that the productivity has risen during this Sprint due to several factors: A successful Sprint Planning with clear tasks for each issue, progress from the backend groups that now allows us to receive Protobuf objects, and lastly, a better understanding of the codebase which makes it easier to navigate in the code.

In this Sprint, we were challenged by the amount of branches that existed in our repository. Our main branch was missing a lot of features that existed on other branches, and we would therefore start to branch out from other branches than main. This should be avoided, and it should be a focus point to get important features into main, such that the branch does not fall behind. Furthermore, we should also create a new branch for each issue, such that each branch only solves one issue.

During the Sprint Planning, we underestimated the amount of time it would take to merge open pull requests. We received a lot of external comments on the pull request regarding the refactoring of the backend communication. We should have assigned more people to this issue, since resolving all the comments required a lot of time. Looking back at it, it would have been helpful to have better internal reviews before sending out the pull request for review.

Lastly, for issue #89, we were not able to implement the reachability check from the

current state within this Sprint. This was due to lacking understanding of how state information was retrieved. However, some group members had this information, so better sharing of both knowledge and blockers could have helped us to solve the entire issue within this Sprint. We will make sure to use the Daily Scrum meetings for discussing blockers in the next, and last Sprint, as we had forgotten to perform this task in earlier Sprints.

9 | Sprint 6: Simulation and Reachability II

Duration: 28/11-2022 - 09/12-2022

9.1 Sprint Planning

This Sprint was the final Sprint of development on ECDAR, and a deadline for pull requests had been set to 07/12. On this date, all pull requests had to be ready, so that reviewing them could commence. This Sprint, we agreed with the group working on multithreading to introduce them to the GUI, as they should start implementing multithreading within the front end.

At the SoS meeting, we agreed that all teams should finish up implementations and focus on the refinement of everything implemented since the start of development. The writing committee agreed to start up on the common report again, as multiple topics had yet to be addressed. Here, it was decided that our group was supposed to review all new parts written within the common report. A date for the last edition of the common report was set to 16/12 to ensure enough time to make a final review of the common report and to incorporate it in the individual group reports.

From our Sprint Planning, the main goal was to implement all simulation and reachability functionality that had yet to be completed. This included, among others, adding the clock values to the simulation trace and sending a reachability request from the current state. Furthermore, testing of these was of priority, as we were ending development on 07/12.

Lastly, we discussed how we in previous Sprints had forgotten to discuss external risks. We therefore argued that for this Sprint, our biggest external risk for the Sprint would be our deadline approaching. This meant that the tasks such as getting our pull requests reviewed could be hindered by the fact that all the other teams were also focusing on their deadline. We would try to prevent this by creating a new branch from the main into

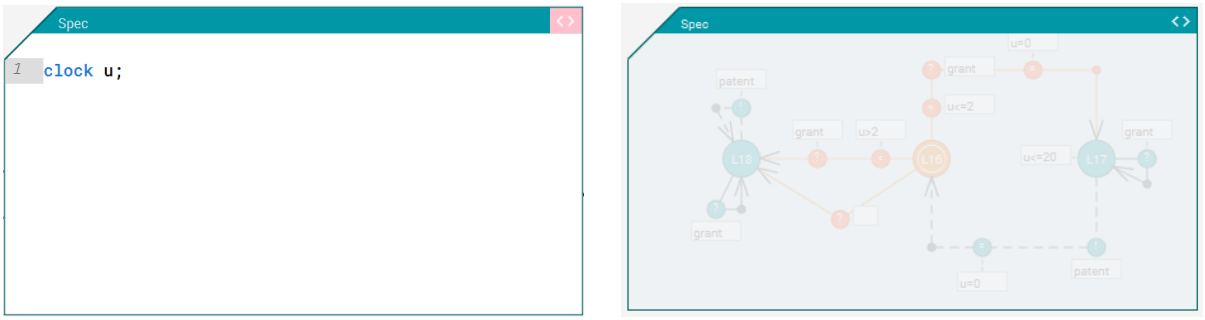
which we would merge all our pending pull requests to only burden the other teams with one combined pull request in the end of the Sprint, where two days had been assigned to reviewing pull requests. Though this will be a comprehensive pull request, it is our hope that it is easier to review one pull request in the assigned review time, rather than multiple small ones while the other groups are still working on their own projects.

9.2 Implementation

This section describes how the issues were implemented. Again, the issues are ordered with simulation first followed by reachability. Finally, the implementation of a more general issue is described.

9.2.1 Issue #103 – Add Clock Declarations to Simulation View

Issue description: In the editor view, the clock declarations can be seen when the user clicks in the top right corner of a component, as seen in Figure 9.1. This feature should also be accessible in the simulation view.



(a) Editor: Clock declarations can be shown by clicking the icon in the top right corner.

(b) Simulator: The same component shows no clock declarations.

Figure 9.1. Comparison of clock declaration display in editor view and simulation view.

Definition of done: The components must have the same representation of clocks in the simulation view as in the editor view, when the "show clocks"-button is clicked.

Implementation description: The problem was found to be in the `ProcessController` class. Even though the declarations of the clocks existed in the cloned components in the simulation view, the simulation view´s `ProcessController` never initialized the declaration text areas for the right corner function. The solution was simply to copy the editor view's functionality to initialize the declared clock variables over to the simulation view.

9.2.2 Issue #102 – Add Clock Values to Simulation Trace

Issue description: Within the trace log of the simulation view, a representation of the clock constraints should be visible to help the user understand the changes in clocks when stepping between edges. This would also result in the user getting the full information about the states by simply referring to the trace log.

Definition of done: Clock values returned from the backend in each simulation step are shown in the trace log.

Implementation description: Within the simulation state, the clock is found within the conjunctions constraints, as seen in Snippet 9.1, line 2. Each constraint has the following properties: x and y which are clocks, a constant c, and a boolean *strict*. Altogether, this represents the constraint as:

$$\texttt{if (!strict) \{ x - y} \leq \texttt{ c \} else \{ x - y < c \}}$$

For each constraint, we extract each property from line 3 to line 6 and on line 7, they are used to create a string that represents that constraint.

```
1 StringBuilder clocks = new StringBuilder();
2 for (var constraint : state.getState().getFederation().getDisjunction()
      ↪ .getConjunctions(0).getConstraintsList()) {
3     var x = constraint.getX().getClockName();
4     var y = constraint.getY().getClockName();
5     var c = constraint.getC();
6     var strict = constraint.getStrict();
7     clocks.append(x).append(" - ").append(y).append(strict ? " < " : " <=
          ↪ ").append(c).append("\n");
8 }
9 return title.toString() + clocks.toString();
```

Snippet 9.1. Excerpt of the `Tracestring` method in `TracePaneElementController.java` that generates a string containing information about locations and clock constraints.

9.2.3 Issue #100 – Handle Simulation Ambiguity

Issue description: When the user sends a step request in the simulation, taking one specific transition in one component, it might force other components to leave the location they are currently in, e.g., due to an invariant on the location or due to the input/output relations. There can be cases where it is unclear from the system itself which transitions

should be taken in the other components if several transitions from the same location give the same result regarding the ability to perform the transition the user requested. An example of such a case is given in Figure 9.2.

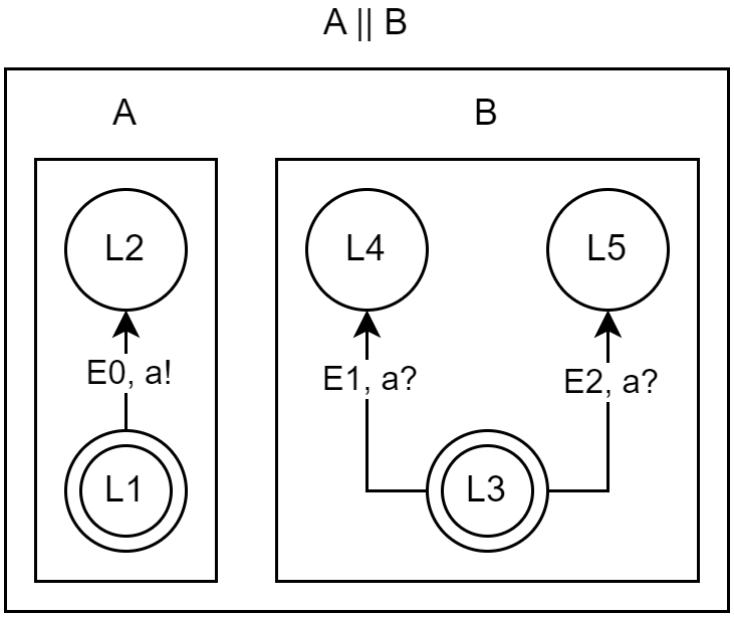


Figure 9.2. An example of a system, where taking transition E0 in component A forces a transition to happen in component B. However, it is not clear from the system alone, which transition should happen in component B. Therefore, this is an example of the ambiguity that can occur in systems during simulation. This figure was provided by the simulation group.

Definition of done: The backend engine returns all the possible states that the user could go into by taking a specific transition in one component. These are returned in a list. This issue is solved when all locations and selectable transitions from all the returned states are highlighted in the components. This will result in a situation where the system appears to be in multiple states at the same time in a non-deterministic manner.

Implementation description: Due to time limitations, this issue has not been solved as was the plan. Instead, the first state in the response from the backend is simply shown to the user.

9.2.4 Issue #101 – Tests for Simulation

Issue description: We would like to see more UI testing done within the simulator, as there is only one integration test right now.

Definition of done: This issue is solved, when a UI test has been made to test the simulation view.

Implementation description: The test uses the TestFX library to define FxRobots, which can simulate user interaction, and to verify the expected state of JavaFX scenes and nodes. The library also provides the method `WaitForFxEvent`, which is used every time the robot should wait for something to happen on the GUI, such as loading a scene or playing an animation.

The purpose of this test is to assess the behavior of the GUI whenever the user attempts to enter the simulation view with and without queries. First, `setUpForTest` is called in line 3 in Snippet 9.2, which creates an Ecdar project with one component and no queries. Then, on line 5, the method `clickOn` is used to simulate a click on the button that switches to the simulator. Because the project contains no queries, we do not want to show the simulation initialization dialog (shown in Figure 9.3) so `assertFalse` is used to test that it is still hidden. For the next part of the test, a query is added to the project on line 11 and the button for entering simulation view is clicked again. This time it should be open so we can select a query and use the FxRobot to click the "Start Simulation"-button shown in Figure 9.3. Then we assert whether the current mode is set to `Simulator`.

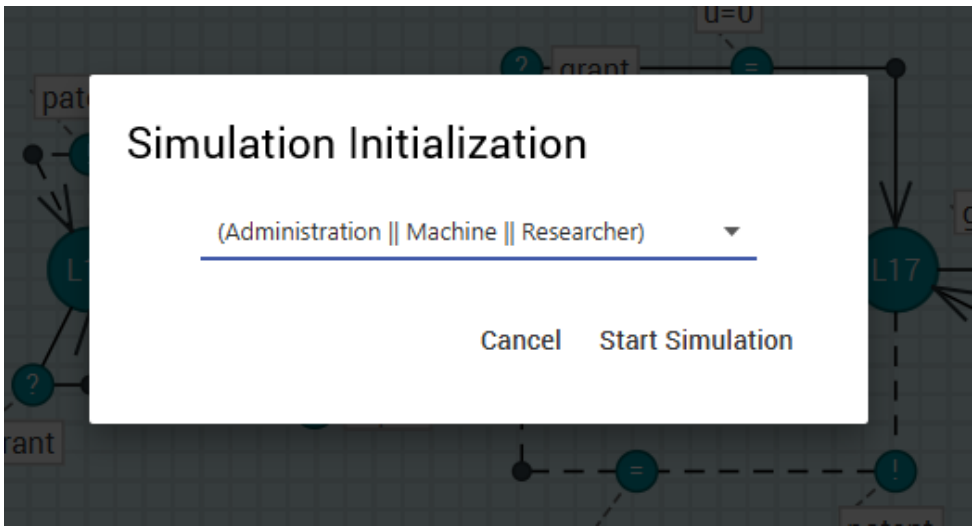


Figure 9.3. The simulation initialization dialog where the user can select which composition to simulate.

```
1 @Test
2 public void UiSwitcherNoComponentsTest() {
3     Ecdar.setUpForTest();
4     WaitForAsyncUtils.waitForFxEvents();
5     clickOn("#switchGuiView");
6     WaitForAsyncUtils.waitForFxEvents();
7     JFXComboBox<String> simDialog =
          ↪ lookup("#simulationInitializationDialog").query();
8     WaitForAsyncUtils.waitForFxEvents();
9     assertFalse(simDialog.isShowing());
```

```
10      ...
11      Platform.runLater(() -> Ecdar.getProject().getQueries().add(new
            ↪ Query("(Component1)", "null", QueryState.UNKNOWN)));
12      ...
13      assertEquals(EcdarController.currentMode.get(), Mode.Simulator);
14  }
```

Snippet 9.2. Code snippet of UiSwitcher test class that simulates user interaction.

9.2.5 Issue #98 – Send Reachability Request with Simulation Query

Issue description: When the first iteration of the reachability check was implemented, the involved class did not have access to the simulation query on which the reachability check should be based. Therefore, in the reachability request, a query was put together based on the names of the components in the simulation, which could indeed be accessed. This was, however, an insufficient solution that imposed a high risk of checking for reachability in a system different from what the user would expect based on the simulation. The query used to launch the simulation should be passed on to and included directly in the reachability request.

Definition of done: This issue is solved when the reachability request is sent with the simulation query as the system definition.

Implementation description: To be able to send the simulation query as the query in the reachability request, the simulation query should be made accessible for the `initializeDropDown()` method in the `SimLocationController` class, which initiates the reachability check. Therefore, a new string variable was defined in the `SimulatorController` class called `simulationQuery`, along with corresponding get and set methods.

The `setSimulationData()` method can be seen in Snippet 9.3 and is located in the `SimulationInitializationDialogController` class and was introduced when solving issue #50 (Subsection 7.2.3) to extract the components in the simulation. To solve this issue, line 2 was added to also retrieve the information about the simulation query.

The `setSimulationData()` method is triggered when clicking on the start simulation button, where it first assigns the simulation query by calling `setSimulationQuery()` with the selected item from the simulation combo box as the parameter. It then uses the

`getSimulationQuery()` in line 6 to retrieve the query to filter out all components, as described in Subsection 7.2.3.

```
1 public void setSimulationData(){
2     SimulatorController.setSimulationQuery(simulationComboBox.getSelectionModel()
        ↪ .getSelectedItem());
3
4     ListOfComponents.clear();
5     Pattern pattern = Pattern.compile("([\\w]*)", Pattern.CASE_INSENSITIVE);
6     Matcher matcher = pattern.matcher(SimulatorController.getSimulationQuery());
7     List<String> listOfComponentsToSimulate = new ArrayList<>();
8
9     while(matcher.find()){
10        if(matcher.group().length() != 0) {
11            listOfComponentsToSimulate.add(matcher.group());
12        }
13    }
14    ListOfComponents = listOfComponentsToSimulate;
15 }
```

Snippet 9.3. A method that sets the simulation query string and filter any unused components from the simulator.

Thereby, the simulation query has been made accessible to the `SimLocationController` class, and it is now used as the system definition in the reachability request instead of the aforementioned composed string.

9.2.6 Issue #92 – Send Reachability Request from Current State

Issue description: In the simulation view, from which reachability checks can be made, it is possible for the user to step through the models, thereby changing the current state of the simulation. As it would be useful for the users to be able to investigate, e.g., dead ends in their systems, the users should have the ability to make a reachability check to a location from the *current* state of the system in the simulation.

Definition of done: This issue is solved when the user can right-click a location in the simulation view and, in addition to the current option of performing a reachability check from the initial state of the system, can select the option of performing a reachability check from the current state.

Implementation description: To solve this issue, we had to add a new option to the

right-click drop-down menu on the locations in the simulation view and we had to update the `getLocationReachableQuery()` method in the `BackendHelper` class to be able to include start states in the request as well. As we were very concerned with not affecting other calls to the method, we made an overload of the `getLocationReachableQuery()` method with an extra parameter: the start state. This overload can be seen in Snippet 9.4 from line 4 to 11. The logic from the original method in lines 1 to 3, which is described in issue #89 (Subsection 8.2.5), was moved to the body of the new overload, and lines 6-9 were added to the existing code. This way, the old method could just call the new method with the value `null` for the start state argument, as seen in line 2, without causing any trouble for the remaining usages.

```
1 public static String getLocationReachableQuery(final Location endLocation, final
      ↪ Component component, final String query) {
2     return getLocationReachableQuery(endLocation, component, query, null);
3 }
4 public static String getLocationReachableQuery(final Location endLocation, final
      ↪ Component component, final String query, final SimulationState startState) {
5     ...
6     if (startState != null){
7         stringBuilder.append(getStartStateString(startState));
8         stringBuilder.append(";");
9     }
10    ...
11 }
```

Snippet 9.4. Code snippet from BackendHelper.java. The body of the overload in lines 4-11 is not shown in its full form, as the code is already shown in Snippet 8.5.

When extracting the state information from the `startState` argument and fitting it to the syntax required by the backend, we realized that we could not include the clock values. This was due to the way we received them from the simulation backend, as the provided format could not be digested by the reachability check. Therefore, the functionality to perform a reachability check from the *current set of locations* was merged into the main branch, and a new separate issue was made regarding reachability check from *current state* – including clock values. The new issue was marked "blocked", as we were not able to implement it without external inputs.

Finally, we added another option to the drop-down menu, which, when pressed, would call the new overload with the current state as an argument. This was retrieved from a `SimulationHandler` object.

9.2.7 Issue #99 – Show Reachability Response Paths on Components

Issue description: When the user performs a reachability check, the engine returns a set of paths – one for each component in the query. Each path contains a list of transitions to take in order to reach the desired state. In addition to informing the user whether the query was successful in the pop-up mentioned in Subsection 8.2.6, it would also be informative for the user to know how the target state was reached. Therefore the user should be informed of the transitions taken.

Definition of done: This issue is solved when the taken transitions taken are highlighted after a reachability check where the target could indeed be reached.

Implementation description: To solve this issue, we had to make some additions to the way the reachability responses were handled. The responses from the backend are handled in the `QueryHandler` class, so the code shown in Snippet 9.5 was added to the case where the response was a successful reachability response where the investigated location could be reached. In lines 2-4, the ids of the taken transitions (edges) are extracted from the received Protobuf message and they are added to the local variable `edgeIds`. Then, in line 7, the `highlightReachabilityEdges()` method is called on the `edgeIds` to highlight them.

```
1 ArrayList<String> edgeIds = new ArrayList<>();
2 for(var pathsList : value.getReachability().getComponentPathsList()){
3   for(var id : pathsList.getEdgeIdsList().toArray()) {
4       edgeIds.add(id.toString());
5   }
6 }
7 Ecdar.getSimulationHandler().highlightReachabilityEdges(edgeIds);
```

Snippet 9.5. Code snippet from QueryHelper.java.

The functionality to highlight the edges was implemented in the `SimulationHandler` class, as they should be highlighted in the simulation view. The method is shown in Snippet 9.6. First, all components are reset to not be highlighted in lines 2-6. This is done to remove highlighting from previous reachability checks. Then in lines 7-11 the edges that are in a returned path are all highlighted. This happens by setting their `isHighlighted` property to which an event listener is attached.

```
1 public void highlightReachabilityEdges(ArrayList<String> ids){
2     for(var comp : simulationComponents){
3         for(var edge : comp.getEdges()){
```

```
4            edge.setIsHighlighted(false);
5        }
6    }
7    for(var comp : simulationComponents){
8        for(var edge : comp.getEdges()){
9            for(var id : ids){
10               if(edge.getId().equals(id)){
11                   edge.setIsHighlighted(true);
12 ...
```

Snippet 9.6. Code snippet from SimulationHandler.java.

This solved the issue. However, it was inconvenient for the user that the edges, at this point in the development, were highlighted with the same color as the possible transitions in the simulation. Also, this solution removed the highlighting of the possible transitions, making it difficult for the user to continue working with the model. Therefore, the development of this feature was continued in issue #123 as described in Subsection 9.2.8.

9.2.8 Issue #123 – Improve Highlighting of Reachability

Issue description: Currently, when checking if a location is reachable, the edges in the response will be highlighted in the same orange color as the edges highlighted from the simulation. It is therefore not possible to distinguish between the edges highlighted from reachability and the edges highlighted from simulation.

Definition of done: The issue is solved when a reachability response results in the edges being highlighted in a purple color instead of orange, thereby making it possible to distinguish between reachability and simulation.

Implementation description: When highlighting the edges based on a reachability response, the property called `IsHighlightedProperty` is set to true and an event listener will make sure that the edge is highlighted. To be able to highlight the reachability result with a different color, a new property called `IsHighlightedForReachability` with a getter and a setter has been made. An event listener has also been added to `IsHighlightedForReachability` which calls the new `highlightSpecialColor()` method. This method will invoke the method called `highlightPurple()`, which has been created on the `Highlightable` interface, where the original highlight method is also located. Several classes implement this interface and as not all of them should be able to be highlighted in purple, the default implementation for `highlightPurple()` is the implementation of the original highlight method. For the relevant classes like `Link` and

`SimpleArrowHead`, an implementation of `highlightPurple()` has been made. This will make sure that both the edges and their associated arrows will be highlighted in purple. An example of an implementation is the `highlightPurple()` implementation in the `Link` class, shown in Snippet 9.7, which highlights the *link* between two locations, i.e., the edge.

```
1 @Override
2 public void highlightPurple() {
3     shownLine.setStroke(Color.DEEP_PURPLE.getColor(Color.Intensity.I900));
4 }
```

Snippet 9.7. Code snippet from the `Link` class.

In Snippet 9.7, the `shownLine` property is of type `Line` which extends `Shape`, a JavaFX abstract class. This makes it possible to use the `setStroke()` method to highlight the edge in a specific color, and here the color is a deep purple color. Similar implementations have been made on the other relevant classes and these will therefore not be presented in the report.

9.2.9 Issue #107 – Refactor Static Properties

Issue description: In order to avoid the use of static properties, the codebase should be refactored such that the list of components for simulation and the query used for simulation can be accessed without it being through a static property.

This approach is a better design for our codebase, since the the list of components and the query for simulation will then be encapsulated in objects instead of having a global representation. This is preferred, since it follows the concepts of the object-oriented paradigm, and it gives the two variables a tighter scope, which, for example, allows for easier testing of the variables.

Definition of done: This issue is solved when the code has been refactored to no longer contain the static properties `ListOfComponents` and `simulationQuery` in the class `SimulationInitilizationDialogController`. Furthermore, the list of components and the query string should still be accessible in a non-static manner, such as through getter and setter methods.

Implementation description: The static properties `ListOfComponents` and `simulationQuery` have been deleted from `SimulationInitializationDialogController` and `SimulatorController`, respectively. The list of components and the query string for simulation have instead been added as private members of the class `SimulationHandler`. As an example of dependency injection, the classes that depend on the list of components

and the query string now contain an instance of a SimulationHandler as a property, and this is how the two variables are now accessed in a non-static manner.

9.3 Sprint Review

As this Sprint was the last Sprint during the project, the main goal was to finish working on both simulation and reachability. We had therefore divided all the issues regarding these into smaller issues to make sure that we could finish everything in time. During this Sprint we also started working on the common report again.

The PO joined the Sprint Review for the last Sprint, so that we could present the results of the last 3-4 Sprints, which had been centered around simulation and reachability.

In regards to simulation, we finished all issues but one. For now, the simulation on the GUI-side does not take ambiguity into account, as this had a lower priority than the other issues.

For reachability we also managed to solve almost all of the issues. We ran into some problems when implementing reachability from a current state, as the reachability team were not able to receive the clock values in the same way as we received them from the simulation group. The simulation group returns the clock values for each state as arithmetic expressions but the reachability group currently does not support this. We discovered this problem quite late and we therefore had to discard this for now, as the reachability group did not have time to solve it.

Regarding the reachability response from the backend, we sometimes see some unexpected behavior. When the reachability group runs a query in their terminal and not in the GUI, they get a response. However, when we then try to run the exact same query in the GUI, we do not always get the same response. Sometimes, we get a time-out or an internal error from the backend, while they get either that the location can or cannot be reached. We did not have time to solve this, but it should definitely be solved in the future. We therefore expressed this to the PO so that he was aware.

Overall the PO was quite pleased with the results of the simulation and the reachability implementations, and he expressed how he was looking forward to incorporating it in the main ECDAR solution.

9.4 Sprint Retrospective

At the Sprint Retrospective, it was agreed upon that this final Sprint has been the most productive Sprint of them all. There are several factors that have contributed to this. First, the backend groups were done or close to done implementing their features, which made it easier for us to implement their changes in the GUI. Furthermore, since the courses had finished, they no longer interrupted our Sprint, which made it easier to leverage the Sprint. Lastly, since we had been working on the project for a while, we now had a more clear understanding of ECDAR and the codebase of ECDAR, which made it easier to discuss and develop.

Even though this Sprint has been the most productive, we still encountered a few setbacks.

As mentioned in Section 9.3, we discovered that the response that we got from the simulation group was incompatible with the format of the request that we send to the reachability group. Therefore, it was not possible to properly implement reachability checks from the current state. A discussion on why this incompatibility was discovered this late was brought up. We concluded that none of the groups were really at fault because this was an implementation detail that was hard to discover before the groups were actually done implementing their respective features. However, more detailed presentations of what each group was doing at the mutual Sprint Plannings might have revealed this problem.

As explained in Section 9.1, we wanted to combine all of our branches into one, such that we only had to burden the other groups with one pull request at the deadline. This was to avoid the bottleneck of waiting for the other groups to review each pull request, as our issues were very dependent on each other. Unfortunately, we encountered a few problems with this. Since we were feeling pressure from the deadline approaching, a lot was merged into this branch that had not received thorough internal reviews, which introduced errors to the new branch. Fixing this required a lot of time from several group members, since we had to backtrack each merge until it was discovered which merge broke the branch. In hindsight, we should have merged one branch at a time into the combined branch, and then tested if the program still ran after each merge.

10 | Post Sprint

After the six Sprints had terminated, the development and the work on ECDAR had come to an end. Though, to ensure a good handover to future students, we decided to set time aside to document all the issues that we had come across throughout the development of ECDAR. Hopefully, this will help future developers.

We created the issues in GitHub on the main branch of ECDAR, from which future developers will most likely start their work on ECDAR. The list of known issues can be found on Ecdar Github, which among others contains the following issues:

- Remove the delay field and relocate the reset button. #118
- Trace log is ordered wrong, as the newest trace is at the bottom. #115
- Scaling is not accurate when opening ECDAR. #112

See the GitHub for the full list.

From these issues, future students will be able to get an overview of what has yet to be completed, as our report primarily documents what has been done. The purpose of these issues is to document our observations and the ideas we had yet to implement. Therefore, each issue contains a small description of the issue, a screenshot if available, and a small suggestion on how the issue could be solved, in the case where we investigated the issue but did not have the time to implement a solution. Some of these issues are also further detailed in Section 14.2

11 | Reflections on High-Level Collaboration

The following section and its subsections are partly written in collaboration with the other ECDAR project groups.

This chapter discusses how the collaboration between the six groups went. To evaluate this, a questionnaire was made, which all the groups had to answer.

To examine the collaboration between groups, focus was on the SoS structure, the effect of making different committees, the DoD and reviewing process, and finally the coordination of dependencies. These matters are discussed in this chapter.

11.1 Scrum of Scrums

SoS was overall well received, however some negative aspects were also present. Most groups agree that it was good and well structured, since it gave an overview of what the other groups were doing, and what their blockers were. This could then be used to see how their own code could potentially be blocked by what another group was doing.

At the SoS meetings, everyone was informed about what had been done and what was going to be done in the different Sprints. The meetings were also used for common Sprint Reviews and Sprint Retrospectives. The Reviews primarily served as an update to the other groups, while the Retrospectives provided both inspiration to other groups through experience sharing, and the communication and teamwork between groups were also discussed. This solved some issues, but the outcomes of the Sprint Retrospectives were not always implemented sufficiently in the practice of the groups.

The length of the Sprints was fixed at the first SoS meeting. It was decided that all groups' Sprints should be two weeks long. This did not seem very agile, since it did not

allow for solving tasks of varying lengths, even if that was needed – but the decision was made in order to align the workflow of all groups, which worked well for most groups.

Overall, the groups were content with the structure of the SoS meetings.

11.2 Committees

One way collaboration was done was through the use of committees. The committees consisted of one representative from each group. Two committees were established: a report writing committee and a gRPC committee.

11.2.1 Report Writing Committee

The report writing committee's job was to decide what to include in the common report part and make sure it was all written. Some of the problems stated by the different groups about the first report writing part were the following: lack of planning, unclear responsibilities, uneven contribution and distribution of work, lacking structure for working multiple people in the same document, missed deadlines, and finally below-expectations efforts.

The report writing committee was reestablished later on to make the last part of the common report. The last part included theory, discussion, and finalizing of sections from the first common report part. Bearing in mind the experiences from the first cycle of common report writing, efforts were put into improving the process.

The second round of common report writing started with a meeting deciding what needed to be added and changed in the report, and dividing that to the different groups so everyone had something to do. Then another meeting was held making sure everyone had written their part and reviewed what the group, with the group number one higher than themselves, had written. Assigning reviews like this meant everyone had to review some specific part and not read the entire report. Since there was a lot less to review, and the group had the full responsibility for reviewing that part, then people were more motivated to review their parts. Overall, the second round went a lot better, since it was more structured and deadlines were adhered to.

11.2.2 gRPC Committee

The gRPC committee's job was to decide what to include in the protocol. All work involving the gRPC committee went well. It was well structured, everyone agreed on the structure, and when something had to be changed or added, it was done quickly and precisely.

The workflow was that whenever a group wanted to add something, it was written by the gRPC committee member of that group, and then reviewed by the remaining members of the committee. The work was well-coordinated, so only one person at a time worked on the protocol, which led to less confusion and conflicts.

11.3 Definition of Done

A DoD was made, but it varied between each group how much it was used and followed. One group used and considered it a great deal, two did a moderate amount, one a little, and lastly, two groups did not use it at all.

For certain groups it felt irrelevant, while other groups felt it was a good idea. Some groups thought that certain parts of the DoD, such as benchmarking all features, were a bit too much to do every time a feature was made, no matter how big or small it was. Multiple groups also thought that it did not contribute much, since it was not strict enough, because not using it had no consequences, and at that point it felt unnecessary to even have a DoD.

In conclusion, the DoD was not used that much. To fix this, if a DoD was wanted, it could be possible to say that code could not be merged if it did not fulfill the DoD. This way, all groups would be forced to follow it and consider it while coding.

11.4 External Review Process

Whenever code had to be merged into the main branches, it had to be reviewed and accepted by at least two members of other groups. The groups are split about how well this went. Half the groups thought it went well, and the other half thought it was less of a success. Most groups agree that the idea was good, but all groups agree that the execution was not ideal.

The problems that are mentioned by almost every group, are that the process typically took a while and it was often the same people doing the reviews. This meant that it was often the same few groups that reviewed all the pull requests, and when they then had to have a pull request approved, it was slow, since they were the ones typically reviewing all the pull requests.

Another problem was that the group dedicated to the GUI did not know Rust (the language of the Reveaal engine). Therefore, they were not capable of reviewing the work of the other groups. Also, when this group made a pull request, it was written in Java, and not in Rust, so the other groups were sometimes slow to review it.

Some suggestions were made by the groups to fix these problems in future projects. These suggestions included appointing a review responsible person from each group at each SoS meeting, making a dedicated time slot where everyone reviewed pull requests, and assigning certain pull requests to relevant groups, if there were any.

11.5 Coordination of Code Dependencies

Coordination of code was about coordinating who writes what, where, and when. It aimed to assure that no two groups were waiting on each other or working on the same code at the same time, thus creating merge conflicts. However, this did not go smoothly.

Multiple times throughout the project, one group was dependent on code from another group in order to progress and was thus stuck. One example of this was between the reachability group and the simulation group, where the reachability group had nothing to do at all because they were waiting for the simulation group to add something that the simulation group had only assigned one person to. The reachability group took over this task the following Sprint, since it had not been completed.

One way to fix this could be by making it clear doing the SoS meetings, who had code that depended on the work of other groups. Then the tasks could be divided between the groups depending on each other, or outright assigned to one of them, if they had nothing else they could do, while the other group had something to work on.

With this, the reflections on the high-level collaboration between the six groups are complete. From this, it can be concluded that the collaboration in general went well with the use of SoS meetings and different committees. For the common report writing, a remarkable progression was made from the first round to the second. Regarding the use of DoD, the reviewing process, and the coordination of dependencies, it went quite well –

but the process would likely have been improved if a more organized approach had been applied.

12 | Discussion

When we started the project work on this semester, we already knew that many things would be different from what we were used to. Not only did we make new groups, as usual, which in itself imposes the challenge of establishing good teamwork within the group - we also had to work together with several other groups. This was due to the fact that on this semester we were working on a multi-project, where several groups collaborated on the further development of an already existing solution. This brought its own set of challenges, both in terms of collaboration and in terms of being thrown into an already existing solution and having to continue the development of that. Exactly what we encountered, and how we dealt with it, is discussed in this chapter.

12.1 Working on the ECDAR Solution

At the first meeting between all groups and the university staff developing ECDAR, we were encouraged to choose either to work on the frontend, i.e., the GUI, or the backend engine. As the reader might have noticed, we decided to work on the frontend. No "goal" was provided from the professors' side, but the GUI was the part of the project where we could immediately see some potential. We were the only group to choose the GUI, while all other groups chose to work on the backend engine.

Choosing to work with the GUI imposed some challenges of different sorts:

- Working in a new programming language (Java) and framework (JavaFX)
- Understanding an existing solution containing equal amounts of missing implementations and unused code
- Keeping up with the development of the backend engine

Each of these points will be elaborated on and discussed in the following sections.

12.1.1 Working with Unfamiliar Languages and Frameworks

Regarding the first point in the list above, it did of course give a remarkable overhead that both the Java language and the JavaFX framework were new to us. Luckily, they have some resemblance to C# and other XML-like frameworks, respectively, so the learning process was manageable. In general, the differences in most of the grammar and features were quite subtle, and a quick Google search could help with most problems. There were, of course, some exceptions, e.g., the concepts of *stage* and *scene* in JavaFX, which were strange to us. Judging by the information we found on these concepts, they could have been utilized better in the solution as well, so that made it even more difficult to understand.

To improve the readability of the code, we tried to be very true to the purpose of the concepts and the intent of the language, e.g., by making getter- and setter methods for new properties. Hereby, we were working consistently with the remaining code, and from our experience, the lack of consistency played some role in causing momentary confusion when trying to read the existing code.

12.1.2 Getting into an Existing Solution

The second challenge was working with and developing upon an existing solution. On previous semesters, we have been used to developing applications from scratch, so this was a challenge that we had not encountered in the university environment before. However, some group members had experiences from jobs, so we were very aware of this challenge. To facilitate the familiarization, we therefore decided to start out with solving simple issues, e.g., hiding a list when it was empty, even though it took out time from working on simulation and reachability, which had become our main goal.

As we got to know the solution better, we also realized that it was characterized by having remarkable amounts of both unused code and "TODO"s, where implementations were missing. Due to the JavaFX framework, it was sometimes difficult for the integrated development environment that we were working in to detect usages. This, combined with the amount of unused code, gave quite some frustrations as making changes to code was not reflected in the GUI when running the program – because the changes were made in code that actually was not used. We learned from the ECDAR group that this was sometimes caused by having the GUI prepare for future development in the backend. Though we understand why this was done, we definitely do not recommend merging unused code into the main branch of a project. It would have been way easier to navigate

if the GUI had been cut to the bone. It is an issue in itself to clean up the code.

Looking back, we all agree that we had good use of this familiarization, where the group members gained some knowledge of different parts of the solution. This provided a stronger foundation for diving into the more complex tasks that were required in order to implement the simulation view, its functionality, and the reachability checks.

One thing we will take into consideration the next time is to focus more on sharing the knowledge we gain – or more precisely to share our problems along the way in a more detailed way. We had a discussion of this in the retrospective of Sprint 5 (see Section 8.4), where some group members had had the experience of being stuck with their issue - when in reality, other group members that had previously been working on the parts they were trying to link to could have helped out. In the Retrospective, we encouraged all group members to share any blockers to the development to prevent such situations.

It is our impression and strong belief that sharing blockers and presenting solution ideas in general is the easiest way to facilitate relevant information sharing. Therefore, in the next project, we will focus more on creating a good culture regarding blocker- and progress sharing at the Daily Scrum meetings.

12.1.3 Having Different Developers on the Engine and the GUI

The last challenge was to keep up with the development of the backend engine. This issue occurred as we were asked to divide the six groups working on ECDAR into frontend and backend groups. We landed on a one-to-five distribution, with only one group working on the frontend. In retrospect, this way of distributing the groups was not ideal. This was especially clear regarding our work on the simulation and reachability features.

Through the development of the ECDAR frontend, we often felt that the order of development was wrong, as we developed frontend features that were not yet supported by the backend, as was the case for the implementation of the simulation feature. A result of this was several reiterations of the way the features should communicate with the backend that was being developed concurrently by another group. Furthermore, the backend engines generally ended up being more powerful than the frontend, with functionalities that could not be accessed through the GUI.

To account for the issues arising from developing the frontend simultaneously with the backend development of the same features, we ended up creating dummy data for testing the frontend. This meant that a lot of time was spent on communicating with other teams to find a common data format that the functionality should use, but even if it felt like

an extra burden at the time, it was a discussion that we would have had to have later on anyway. Nonetheless, the process felt nonoptimal, and the problems made us discuss what should come first: front or backend development. This led us to the following reasoning of whether to start with one or the other.

Reasons to build the frontend first:

- It allows you to see the visual aspects of the project and get a better sense of how it will look and feel. This could be extra helpful for design-focused development where the user interface is a key part of the overall product.
- It can help identify potential issues or challenges with the user interface early in the development process, allowing them to be addressed before they become more difficult to fix in the backend.

Overall, building the frontend first can be a good approach for projects that prioritize the user interface, collaboration, and early feedback from stakeholders.

Reasons to build the backend first:

- It allows focusing on the data and logic of the project without being distracted by the user interface. This could be extra helpful for projects where the backend is complex or requires a lot of planning and development.
- It could help to establish a solid foundation for the project, by defining structures and data that will be used by the frontend. This could make it easier to integrate the frontend into the overall project later on.
- It could help to test and iterate on the backend functionality without worrying about the user interface, as the interface is more flexible. This could be especially useful for projects with a high degree of uncertainty or where the backend is the important part of the project.

Overall, building the backend first can be a good approach for projects that prioritize the data and logic of the project, i.e., a solid foundation.

With these thoughts in mind and the fact that ECDAR is computationally heavy, we found it would have been more logical to have backend teams control when features should be developed in the frontend, i.e., by developing the backend engine first - as long as we insist to separate our teams so that they only work on frontend or on backend. We, however, argue that for future development it could be beneficial to have backend teams explain their vision of features, before starting frontend development of said features.

The approach described above would however not have solved the issue we saw with the reachability feature, as described in Section 9.4. Here, we realized very late in the process

that the clock values contained in the simulation step response could not be parsed by the reachability feature in the backend. At the time of discovery, the reachability group had terminated their development process, and so the issue was not fixed. The issue arose due to different factors, amongst others that the backend group working on simulation had a more "waterfall" like approach and only released their work in the end of the project period. Nonetheless, it gives an indication of why developing a backend feature *completely* before developing the frontend might not be the best solution either.

Instead, we could consider avoiding the frontend/backend split and split groups purely based on feature. As we were only one frontend group to serve five backend groups, some backend groups ended up implementing their backend features in the frontend themselves. Despite some initial frustrations with familiarization with the frontend code base, this worked well – so we can only imagine how the development process could have been if all groups had been working feature-based on the frontend and backend simultaneously. At least, we expect that it would have increased the coherence of the front- and backend in terms of feature functionality and communication of information.

To sum up, we recommend for future projects to either let the backend development drive the frontend development, or to ensure that the same groups are working on both aspects of the solution. These approaches would hopefully counter the issues we experienced of 1) trying to develop the frontend without being able to communicate with the backend, 2) realizing that the backend was much more powerful than what the GUI was designed for, and 3) discovering any incompatibilities too late.

12.2 Evaluation of Our Solution to the Problem

When we started working on ECDAR, there was no particular goal for what we had to achieve in the end. Already during the first Sprint we tried to set a goal for what features we wanted to develop during the project process. The goal was to implement simulation, meaning that it should be possible to use the GUI to simulate through components. Furthermore, it should be possible to send a reachability request to the backend and the response should then be visible in the GUI. These two were the major features that we implemented during all the Sprints. In addition to this, several smaller issues were solved.

12.2.1 Evaluation of Simulation and Reachability

A lot has been achieved during the Sprints regarding simulation. It is now possible to select between the editor view and the simulator view. When first initializing the simulator view, a drop-down is displayed where a query has to be selected for simulation. When a query has been selected, the simulator view is displayed with the components that are appropriate for the selected query.

In the simulation view, seen in Figure 12.1, it is possible to step through components by clicking valid edges. In order to indicate to the user which edges are valid, the valid edges are highlighted in orange. In the left panel of the simulation view, the simulation trace is displayed. The trace shows all the transitions that have been taken during simulation, as well as the clock values at each transition. It is possible to step back and forward between transitions in the simulation trace. It is also possible to reset the simulation.

One flaw that has currently been discovered with simulation is that sometimes taking a transition can lead to ambiguity, as described in Subsection 9.2.3. The proper way to solve this is to show all possible states in the components along with all possible transitions, such that it is up to the user to decide which transition should be taken. However, this is not currently implemented due to time constraints.

In regards to reachability checks, it is possible to do them from the initial state of the components. Furthermore, it is also possible to do reachability checks from the current set of locations without taking the clocks into account. If a location is reachable, the path to that location is highlighted in purple.

It is not possible to do a reachability check from the current location where the clocks are also being considered. This is because of the incompatibility between the backend responses, which is also discussed in Subsection 9.2.6.

The features of the simulation view can be seen in Figure 12.1.

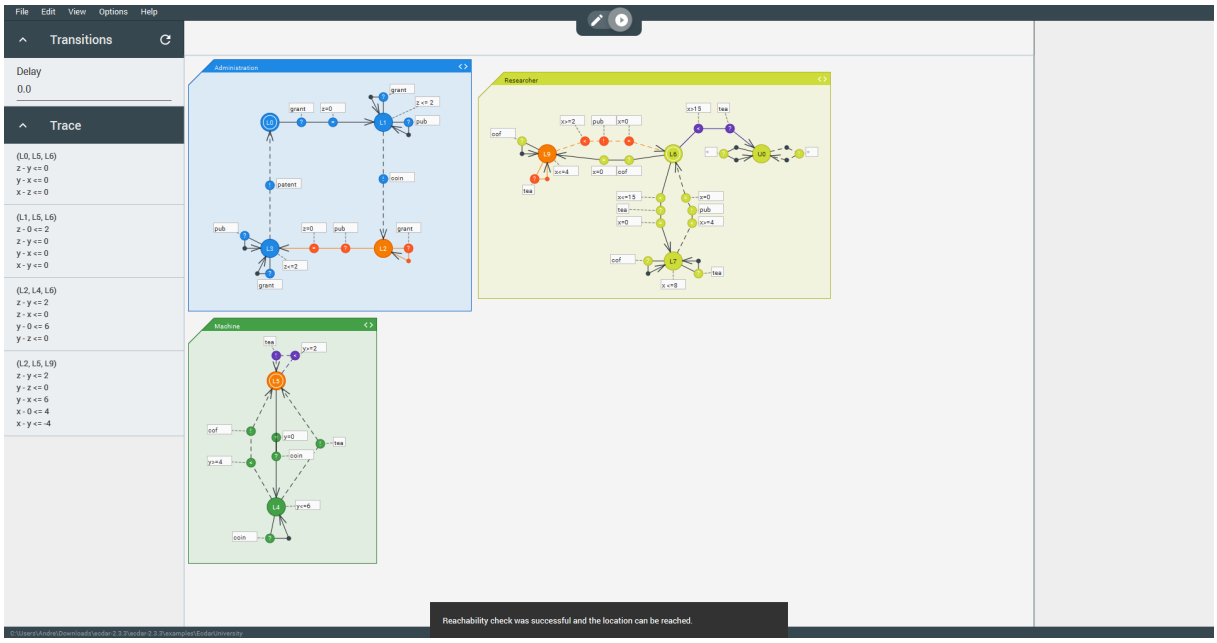


Figure 12.1. The simulation view. Edges highlighted in purple are a result from a reachability check. Orange edges are a result from a simulation step response. In the left panel the clock constraints are presented.

12.2.2 Evaluation of Simulation View

Before we started working on simulation we were handed a lot of old code from a previous master's group. They had already implemented the functionality of switching between the editor view and the simulation view and we therefore chose to continue working on their code. There was a lot of code that was not really being used, which made it quite difficult to just read the code and understand what was going on. Along the way, we therefore tried to delete unused code to make the code more readable. During the last couple of Sprints we noticed that when switching from the editor view to the simulator view, only the contents of the three main panels are replaced. This means that if a panel should not be in the simulator view, it is currently not possible to remove it as it also would be removed from the editor view. This makes it quite difficult to further develop the simulator view due to these limitations. As mentioned in Section 6.2, JavaFX uses a structure where the application is divided into a stage and one or more scenes which are then further divided into different nodes. To make the code more understandable and to follow the structure of JavaFX, the simulator view could be created as a new and individual scene. This would remove the limitations currently present when having the simulator view as a "copy" of the editor view structure.

Overall, there currently might still be a lot of unused code that we have not discovered, which makes the code less readable. This should be removed in the future. Also, it is a

good idea to structure the different parts of the GUI into scenes and nodes to follow the usual structure of JavaFX applications.

12.2.3 Evaluation of Testing

To ensure that the features we have implemented work as they are intended to, it is important to test the code behind the features. For most methods, we have written one or more unit tests to check that their output is correct. Most tests are quite simple unit tests, which are only testing the Java code and not how this is reflected in the GUI. For some tests, we have written different test cases that will be used as input in the tests, but to improve our tests we could randomly generate new inputs when running the tests to make sure that they not only pass with the hardcoded inputs that they currently have.

Our contribution to the ECDAR project has mainly been to create frontend implementations to the features developed by the backend groups, thereby making them accessible through a GUI in addition to the command line interface. As the UX design of ECDAR is still in a very early stage, we have not prioritized determining whether or not our visual implementations are user-friendly and intuitive. Therefore, we have not been able to verify that the visual implementations that we came up with are optimal in regards to usability. When it comes to frontend tests, usability testing is usually an effective test method to see if the layout and workflow are intuitive for the user. We have not performed any usability tests, but to ensure that the features meet the expectations of the ECDAR team this should be done in the future.

Another method for frontend testing is to write tests that will check if certain events occur as expected in the GUI. We have written a GUI test for checking that the simulation start request is sent to the backend, and for checking that the simulator view is entered. Only one test was written, as GUI testing was new to us and we therefore had to get familiar with it before being able to write tests ourselves.

Altogether, frontend testing differs from backend testing. Usability testing is usually a part of frontend testing and to ensure that ECDAR is developed towards the actual goal, this should be done. Testing the code that produces the different views in the GUI is also crucial. This has currently been done by testing units of a feature in a GUI test, and through unit tests of various methods. To make sure that all features behave in the correct way, more GUI tests should be added in the future, to ensure that all features are thoroughly tested.

12.3 Process Evaluation

In this project, we performed agile software development, using the Scrum framework. This was new to most of us. In this section, we discuss our experiences with this work form.

12.3.1 Agile Software Development

As mentioned, this was our first time performing agile software development. Overall, we have had both advantages and disadvantages from applying this method.

The reason for doing agile software development was that it would be able to handle the uncertainties that we were expecting: unfamiliar languages and frameworks; a complex, existing solution; and changing requirements coming from either the PO or as a result of backend groups making changes to data formats etc.

Overall, we agree that the agile work form was well suited, as we were not forced to try to make a plan for the entire semester, which would have been very difficult to do. It allowed us to take the time needed to work on smaller issues in order to familiarize ourselves with the solution as well as its language and frameworks. It also worked well in terms of changing requirements and changing surroundings, e.g., when the reachability group decided to make small adjustments to the structure of the data returned in a reachability response. Due to the limited duration of each Sprint, it was unlikely that we would spend more than two weeks on code that would then have to be changed. This is an advantage compared to the waterfall method.

Speaking of changes to the requirements, customer engagement is weighted higher than contract re-writing in agile software development. We have had continuous contact with the PO, giving us good opportunities to discuss the development. Though we have not missed a formal contract, we did have situations, e.g., related to the development of the simulation view, where we after a meeting with the PO were unsure if we had forgotten parts of the discussions. It would have helped to at least keep some memorandum of the outcomes of the meetings.

Finally, the agile methodology states that working code is more important than comprehensive documentation. This was, as described in Subsection 2.1.1, contradictory to the nature of the student projects, where the report is one of the main products. There is no doubt that the documentation of the Increments has been an interrupting activity that does not go well with agile software development.

Another challenge to the agile software development is the overall structure of the project, which is that multiple teams of students work on the project during one semester. It then gets paused for the next semester, to get picked up by new students on the following semester. Due to this structure, the project as a whole might have benefited from using a waterfall method to improve the project regarding documentation. While Scrum's lack of documentation is often brought up as its downfall, it is however somewhat mitigated by the fact that students are required to write a handover section in their report, which should describe what future students should keep in mind and what to take away from previous work.

12.3.2 Scrum

For the specific agile framework, we have been using Scrum. Scrum is rather well-defined and it is a requirement that all participants are familiar with Scrum when using it. Naturally, this gave quite a learning overhead in the first Sprint, as we spent much time investigating Scrum and trying to understand how to implement its practices. However, when we first got to understand Scrum, we also started seeing the advantages of the framework.

As described in Subsection 2.1.2, Scrum is structured in Sprints, within which the development takes place. It starts out with a Sprint Planning. In the first couple of Sprint Planning meetings, we were being quite superficial, but inspired by one of the other groups we started breaking down our issues into the smallest possible parts, as can be seen in, e.g., Figure 8.1. This helped assessing their workload, and though we probably overcompensated slightly, we see how a balanced level of decomposition is very beneficial.

In the end of each Sprint, a Sprint Review and Sprint Retrospective were held. These meetings also proved to be very helpful, though it often felt like the Sprint Reviews came too soon in the development. This is an observation that could be taken into consideration when deciding on the lengths of the Sprints. We decided from the beginning of the semester that all Sprints should have a duration of two weeks, which probably was not a good way of going around it. Instead, we could have adjusted the length of the Sprints at a SoS meeting, e.g., after the first two Sprints, as we gained experience working with Scrum and ECDAR. Regarding the Sprint Retrospective, this proved to be a very relevant meeting which lead to many improvements of the process – as reflected in this discussion.

Another factor that left our Sprints less productive was the interruptions caused by lectures. When we had many lectures during a Sprint, it meant that not a lot of progress was made. During the Sprints with a lot of scheduled lectures, we could often only set

aside 3-5 working days for the project, and the span of a whole day also had to be set aside for Sprint Planning, Sprint Review, Sprint Retrospective, and supervisor meetings. This means that for some Sprints, we only had a few days to actually implement the Sprint Goal. Based on this, it was clear to see that it is not optimal to work with Scrum and short Sprints when we also had to spend time on attending lectures.

Another activity in the Sprint was the Daily Scrum. Similarly to the Sprint Planning, we started out with rather shallow meetings. We would just say what we had done, and what we were planning on doing, without going into too much detail. This backfired later, when some group members were stuck as they were lacking specific knowledge of the solution that other group members possessed. In hindsight, we discussed that the confusion might have come due to the feeling that the Daily Scrum should be a very short meeting - motivated by the fact that participants are supposed to stand up for the duration of the meeting. However, we learned that the value of the meeting was much higher, if we went into slightly more detail compared to what we did in the beginning of the project.

Finally, we discussed how working with the PO went in relation to Scrum. Working with a PO requires that both the team and the PO are well-trained in agile methodology, as they will have to work together on which issues should be included in the different sprints. While the PO should not directly select the issues, they should point the team in the desired direction and define a Product Goal, whilst it is the team's responsibility to define issues and handle resource management. During the initial stages of the project, we struggled to find the main Product Goal, as we mostly worked on random issues and bugs. However, as the work continued on ECDAR, we felt that developing the simulation feature and implementing reachability checks gave us a sense of direction. In the future, it will be important to define a Product Goal early in the process so that the developers can focus on achieving that goal.

Overall, we have learned much through this project, both in terms of performing agile software development in a larger development environment, and in regards to continuing the development of an already existing solution. In the next chapter, we will give a conclusion on how the solution was improved and what we gained through the implementation of Scrum.

13 | Conclusion

The primary purpose of this project was to extend and improve the functionality of the ECDAR GUI, with a special emphasis on implementing the user interface for performing simulations and reachability checks. These two features, along with other backend engine features, were being developed simultaneously by backend groups. Thus, the secondary purpose of this semester was to coordinate the collaboration between the groups.

In our group, we successfully solved a set of minor GUI issues in the beginning of the project. For the last 3-4 Sprints, we focused on the simulation and reachability functionalities. A simulation view has been implemented, in which the user can simulate a system. The involved components are visualized, and the user can step through the states of the system. The user is provided with an overview of the states that have been visited in a trace, and it is possible to jump back and forth through visited states.

In the simulation view, the user now has the option to request a reachability check to a selected location. The reachability check can be performed from either the initial state or from the current set of locations, and the response is presented to the user.

The simulation and the reachability functionalities have been validated through several unit tests ensuring that the correct requests are sent to the backend. Furthermore, a GUI test has been used to validate that the simulation view is entered correctly. The tests indicate that the solutions work as desired, which was confirmed by a rather content PO.

To coordinate the work in the larger development environment, we used agile software development. We experienced how this prevented stress related to uncertainties. Through the Scrum activities, the teamwork was improved during the semester. Even if a rigid Sprint-structure does not go well with interrupting courses and report writing, we still came to the conclusion that Scrum is a suitable framework for this type of project work.

From this, it can be concluded that the functionality of the ECDAR GUI has been significantly improved as the simulation and reachability functionalities now can be accessed through the GUI.

14 | For Future Students

This chapter is aimed at future students who will be working on ECDAR. It describes what Increments would give the highest value in our opinion. It also gives concrete suggestions for good first issues with descriptions on how solving them would contribute to the students' understanding of the solution as a whole.

14.1 Future work

For future work on the ECDAR GUI, different improvements could be made to both the existing features but also new features could be added. In this section, we describe issues that we find the most important to implement. Furthermore, we will give our suggestions towards good starting issues for future students who should decide to work on the ECDAR GUI branch.

Currently, when performing reachability checks the response is not always accurate, as the backend seems to send a different response than the response we receive in the GUI. Due to the time frame of the project, we did not have time to investigate why this is the case. This should definitely be fixed in the future to make sure that reachability checks always act reliable.

Another improvement could be to remove unused code from the codebase. The codebase is already quite big and the unused code makes it more difficult to get familiar with the code. A clean-up in the code would therefore make it easier for future students to read and understand the code.

As JavaFX applications usually are structured in a scene and stage layout, it would be nice to fully implement this in the GUI as well. This would especially be useful for the entire simulation view. The simulation view is currently closely related to the editor view. It is therefore not possible, e.g., to remove one of the main panels in the simulation view without affecting the editor view. Therefore, it would be good to make sure that the

simulator view can be changed without having to worry about affecting the editor view.

14.2 Good First Issues

In an effort to help future student developers, we want to describe some issues that would be good as their first issues. These should be solvable whilst giving a good insight into how the ECDAR GUI is structured.

At the current time, a good first issue would be issue #115: Trace log is ordered wrong, as the newest trace is at the bottom. Implementing a solution to this issue requires that the developers gain a good understanding of JavaFX and its system of stages and scenes. Furthermore, the issue stems from the simulation view, and therefore the developer has to understand both ECDAR's file structure and its separation between what is in the editor view and the simulation view. Lastly, this issue provides an insight into the way we have implemented functionality in the simulation view, which is a new iteration to ECDAR that was not there at the start of the project.

Another good first issue could be issue #118: Remove the delay field, and relocate the reset button. This issue also concerns the simulation view. Currently, there is a delay field for increasing the clocks which are not being used. This should therefore be removed and the reset button should be relocated to a more intuitive place. When trying to implement a solution to this issue, the developers get an overview of how the FXML files are connected to the Java files, and how methods can be called in the FXML files.

Finally, a good first issue might also be issue #116: In the simulation view, "<=" must be changed to "≤" and ">=" to "≥". Note that this has already been solved in this project, but only in the editor view. A similar fix can be used in the simulation view. The developers could also implement a wider solution, by making sure that the replacement automatically happens every time a user types in any text field. Solving this issue provides the developers information on where different classes are located in the solution. In this case, it is about tags and how they are represented in the GUI.

Bibliography

[1] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*. Cham: Springer International Publishing, 2018, publication Title: Handbook of Model Checking /. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-319-10575-8.pdf

[2] A. David, K. G. Larsen, U. Nyman, A. Legay, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*. United States: Association for Computing Machinery, 2010. [Online]. Available: https://vbn.aau.dk/ws/portalfiles/portal/58768871/HSCC2010cameraready.pdf

[3] M. Goorden, "Introduction Automata TIOA and TIOTS Specifications and implementations Operations Conclusion Specification theory for timed input-output automata: The theory behind ECDAR," Sep. 2022. [Online]. Available: https://www.moodle.aau.dk/mod/resource/view.php?id=1451126

[4] gRPC, "grpc.io," 2022. [Online]. Available: https://grpc.io

[5] Google, "Protocol Buffers." [Online]. Available: https://developers.google.com/protocol-buffers

[6] ECDAR, "ECDAR.NET," 2022, publication Title: https://www.ecdar.net/. [Online]. Available: https://www.ecdar.net/

[7] I. Sommerville, *Software engineering*, 10th ed. Harlow: Pearson, 2016, sommerville.

[8] A. Cline, *Agile development in the real world*. Berkeley, CA: Apress, 2015.

[9] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin,

S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for Agile Software Development," 2001. [Online]. Available: http://www.agilemanifesto.org/

[10] K. Schwaber and J. Sutherland, "Scrum guide | scrum guides," 2022. [Online]. Available: https://scrumguides.org/scrum-guide.html

[11] GitHub, "Fork a repo." [Online]. Available: https://docs.github.com/en/get-started/quickstart/fork-a-repo

[12] "Ecdar," Oct. 2022, original-date: 2022-09-12T08:55:50Z. [Online]. Available: https://github.com/Ecdar-SW5/Ecdar-GUI

[13] "Planning Poker." [Online]. Available: https://www.youtube.com/watch?v=gE7srp2BzoM