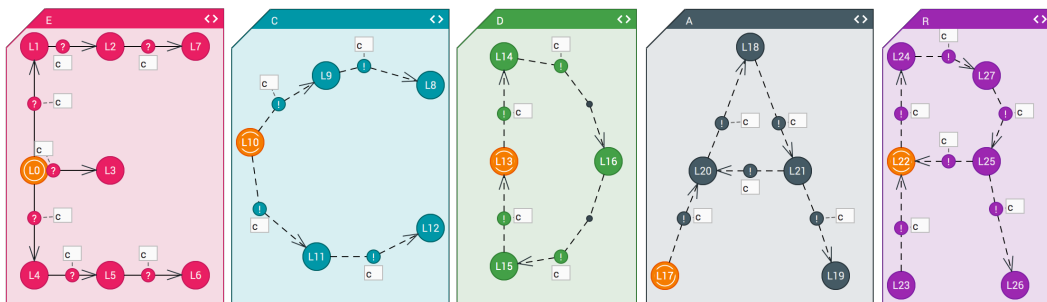


Extending Ecdar 2.0 with a User-Centered Visual Simulator

Master's Thesis



Project Group:
deis105f18

Group Members:
Casper Møller Bartholomæussen
Rene Mejer Lauritsen

Supervisors:
Ulrik Mathias Nyman
Dimitrios Raptis

Summary

In this project we build upon the existing tool ECDAR 2.0, an Integrated Modelling and Verification Environment (IMVE) for the theory of Timed I/O Automata (TIOA). ECDAR 2.0 is the result of the work done by Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1] last semester and it introduces IMVE features and the concept of system views. ECDAR 2.0 is designed to be a new and modern model checker which utilizes the ECDAR 0.10 backend. It is also a hard fork of the H-UPPAAL project¹.

This semester we have turned our focus towards the design and implementation of a visual simulator for ECDAR 2.0. The main challenge of developing a simulator for ECDAR 2.0 is designing the user interface for it since the backend is already available through ECDAR 0.10. We approach this challenge by designing, implementing, and evaluating the simulator in a user-centered manner. To begin this approach we have, in the requirements engineering phase, made a questionnaire with questions about which features are important, based on features we identified in other tools with a visual simulator. Based on the answers from 18 participants, we identified the most important features for a visual simulator.

To assist us in the design phase we adopt a user-centered approach inspired by Buxton [2] and Pugh [3], namely the design funnel. The purpose of the funnel is to sketch concepts and features in phases of *controlled convergence* and *concept generation* i.e. we start by sketching concepts for each *must have* feature, choose some of the sketched concepts, generate new sketches based on the previously chosen sketches, and repeat the process until we end on a final design of the visual simulator.

We went through three design phases of concept generation and convergence. After the three design phases, we settled on the final design which serves as the base for the implementation of the simulator. The final design did undergo changes during the implementation of the simulator. Some examples of changes in the design: A left-hand bar has been introduced instead of using tabs for changing modes, the transition chooser and the trace log have been swapped, an alternative solution for how to show the state values has been selected, and the "eye" feature has been left out.

We assumed that the ECDAR 0.10 backend worked without any major issues, as we have not experienced any problems in regards to queries, but that was not the case. This led us into an extensive reverse engineering process of the communication with the backend, such that we could identify what went wrong when we tried to advance in a simulation. After some segmentation faults, countless consultations with the UPPAAL documentation, and a lot of trial and error, we figured, by coincidence, that calling the same method multiple times returned an expected answer.

To validate the design choices we have made in regards to the simulator, we have conducted a usability evaluation. The evaluation was conducted with nine participants

¹H-UPPAAL GitHub repository: <https://github.com/ulriknyman/H-Uppaal>

where six of them have used a model checker for at least one project, and three of them have more than two years of experience using a model checker. The participants carried out three tasks, which covered all of the *must have* features, in a random order to balance against the carry over effect. In total, 18 usability issues were identified with 9 of them being cosmetic, 6 serious, and 3 critical.

The visual simulator is a great step towards making ECDAR 2.0 a feature complete IMVE, but the vision does not stop here, there is still plenty of room for features and additions for ECDAR 2.0. Some of which could be: An improved text editor for declarations, a new and open engine for ECDAR, fix the critical and frequent usability issues, support for version control systems, and more.



AALBORG UNIVERSITY

STUDENT REPORT

Department of Computer Science

Selma Lagerlöfs Vej 300

DK-9220 Aalborg Ø

<http://www.cs.aau.dk>

Title:

Extending Ecdar 2.0 with a User-Centered Visual Simulator

Theme:

Master's Thesis
Semantics and Verification

Project period:

Spring semester 2018

Project group:

deis105f18

Participants:

Casper Møller Bartholomæussen
Rene Mejer Lauritsen

Supervisors:

Ulrik Mathias Nyman
Dimitrios Raptis

Pages: 57

Date of completion:

June 15, 2018

Abstract:

In this project we extend ECDAR 2.0 with a user-centered visual simulator.

The main challenge of developing a simulator for ECDAR 2.0, is designing the user interface for it. We approach this challenge by designing, implementing, and evaluating the simulator in a user-centered manner.

We inspect the visual simulators of other tools, and use questionnaires to understand the functional requirements for a visual simulator. The requirements are prioritized using the MoSCoW method. We use the *must have* features as a Minimum Viable Product (MVP).

We adopt a user-centered approach, namely the design funnel, where we through three phases in the funnel, sketch multiple concepts for each *must have* feature and evaluate them using different methods.

The final design of the visual simulator has been implemented, and evaluated through a usability evaluation. We identified 18 usability issues, where 9 of them are cosmetic, 6 serious, and 3 critical.

The visual simulator satisfies the MVP, and it complies with the user-centered design, and hereby the main challenge.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the authors.

Preface

This report is the product of a 4th semester project by two Software Engineering master students at Aalborg University. This project is the master's thesis project and contains the work of 2 * 30 ECTS points. The master's thesis is based on the work from the pre-specialization project namely [1].

The reader is expected to possess knowledge of computer science to the same degree as a 4th semester Software Engineering student at the Master level at Aalborg University, including basic knowledge about model checking and verification, in order for the reader to benefit the most from reading this report.

We want to give special thank to our supervisors Ulrik Mathias Nyman and Dimitrios Raptis for helping us during the project period. We also want to thanks the participants of the usability evaluation for helping identify usability issues in ECDAR 2.0, as well as the people answering our questionnaire to find the important features in a simulator.

Reading Guide

The report is written in chronological order and is recommended to be read as such. Personal pronouns refer to the authors of this report. All figures in the report are made by the authors unless stated otherwise in the figure caption.

Citation Style

All references throughout the report are in IEEE style. The bibliography can be found at the end of this report on [page 59](#).

Figures

Figures in this report are centered horizontally. A caption beneath the figure describes the content of the figure, as well as its reference tag. Figure [0.0](#) is an example of this.

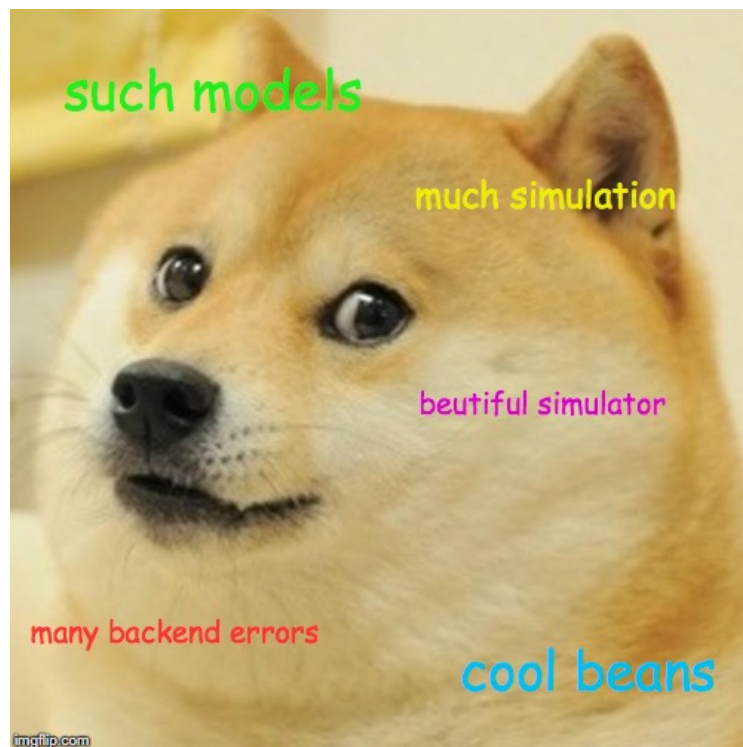


Figure 0.0: This is an example of a figure

Contents

1	Introduction	1
1.1	Method	2
1.2	Requirements Engineering	3
1.3	Related Work	6
2	Design	11
2.1	First Phase	11
2.2	Second Phase	18
2.3	Third Phase	24
3	Implementation	31
3.1	Reverse Engineering the Communication with the Backend	31
3.2	Architecture	32
3.3	Overview of the Simulator	33
3.4	Feature #1 – Manual Stepwise Execution of Simulation	35
3.5	Feature #2 & #4 – Overview of Steps in a Trace & Go Back and Forth in Trace	36
3.6	Feature #3 – Inspection of State Values	37
3.7	Feature #21 – See Traces from Verifier in the Simulator	38
3.8	Feature #6 – Go to Initial State	39
3.9	Summary	40
4	Usability Evaluation	41
4.1	Participants, Setup, and Tasks	41
4.2	Results	43
4.3	Summary	50
5	Discussion and Conclusion	51
6	Future Work	55
6.1	Improved Text Editor for Declarations	55
6.2	Usability Issues	55
6.3	New Engine for Ecdar	55
6.4	Future Work from the Previous Semester	56
	Bibliography	59
	Appendix A Tool Inspection	61
	Appendix B Feature Selection Sheet	65

Appendix C	Sketch Rankings	67
Appendix D	Configuration Sketches	69
Appendix E	Usability Evaluation – Tasks	73
Appendix F	Usability Evaluation - Issues	77
Appendix G	Usability Evaluation – Suggestions	79

1 Introduction

In Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1] we introduce and develop ECDAR 2.0, an Integrated Modelling and Verification Environment (IMVE) for compositional real-time systems. ECDAR 2.0 utilizes the engine of ECDAR 0.10 for verification of models in the Timed I/O Automata (TIOA) formalism. It intends to improve on the modelling process, compared to ECDAR 0.10, by introducing features that prevent invalid models, improve productivity, and provide feedback to the user. It also introduces system views for easy creation of compositional systems and verification queries.

ECDAR 2.0 also benefits from the Integrated Development Environment (IDE) inspired features of H-UPPAAL (developed by Mouritzsen and Jensen [4, 5]), as ECDAR 2.0 is built upon its codebase. ECDAR 2.0 is thus a great environment for verification of compositional systems, but, it is missing a visual simulator in order to make it a more complete IMVE. Besides, visual simulators are also common in other modelling and verification tools e.g. the tools in the UPPAAL family and Petri net tools like CPN Tools.

A visual simulator can be useful in exploring and debugging models. As previously mentioned, ECDAR 2.0 uses the engine of ECDAR 0.10 for verification, and that engine already provides functionality for simulation.

In Mouritzsen and Jensen [5] they considered adding a simulator to H-UPPAAL, and they state the challenges like this: "The challenge with this (ed. the simulator) is not retrieving the trace and states from the verification engine, which is already possible through its interface. Instead the challenge is presenting these states and traces to the user." — Mouritzsen and Jensen [5].

Thus, the main challenge is to take the traces and states from the engine, and present it to the user such that it improves the user experience and productivity when using ECDAR 2.0. Based on our knowledge of the engine, we agree with their assessment of the challenge. The following description describes how we intend to approach the challenge:

We adopt a user-centered approach to the development of a simulator for ECDAR 2.0. This approach should investigate whether there exists requirements for a visual simulator in a model checker, and involve users in order to understand their needs for such a simulator. The design of the simulator should follow an approach that inspires creative thinking and evaluation of ideas. The implementation should be evaluated with participating users to validate the user-centered design and whether it complies with the found requirements.

For the first step in this approach we will, in the following sections, present a method for the design process of the simulator, and explore existing tools to identify which features they provide and how they handle visual simulation.

1.1 Method

We adopt a user-centered approach to the design and development of a visual simulator for ECDAR 2.0. The approach is based on sketching and applying divergent and convergent thinking. We follow this approach to arrive at the *right* design for our presented problem.

The sketching process is inspired by Buxton [2]. Buxton mentions disposable, plentiful, and quick to make as some of the attributes that sketches possess [2]. These attributes make sketches useful in the ideation stage, where we want to explore different types of designs. Later in the design process, we might want to produce prototypes, that can be more specific and refined than the sketches.

The design funnel as proposed by Buxton [2] presents the design process as starting broad, sketching and exploring ideas, and ending narrow, prototyping ideas for usability tests. Figure 1.1 presents a version of the design funnel from Buxton [2], who has based his work on Pugh [3]. This representation has the benefit of alternating between generating and choosing ideas, which is an essential part of the design process. Note that we do not completely follow the method presented in Pugh [3], but mainly the notions of *controlled convergence* and *concept generation*. In Section 1.2 we present our approach to find the functional requirements that can be used as an input to the funnel. In each phase of the funnel, we generate sketches, based on the previous phase or the requirements, and convergence on the concepts using different evaluation methods.

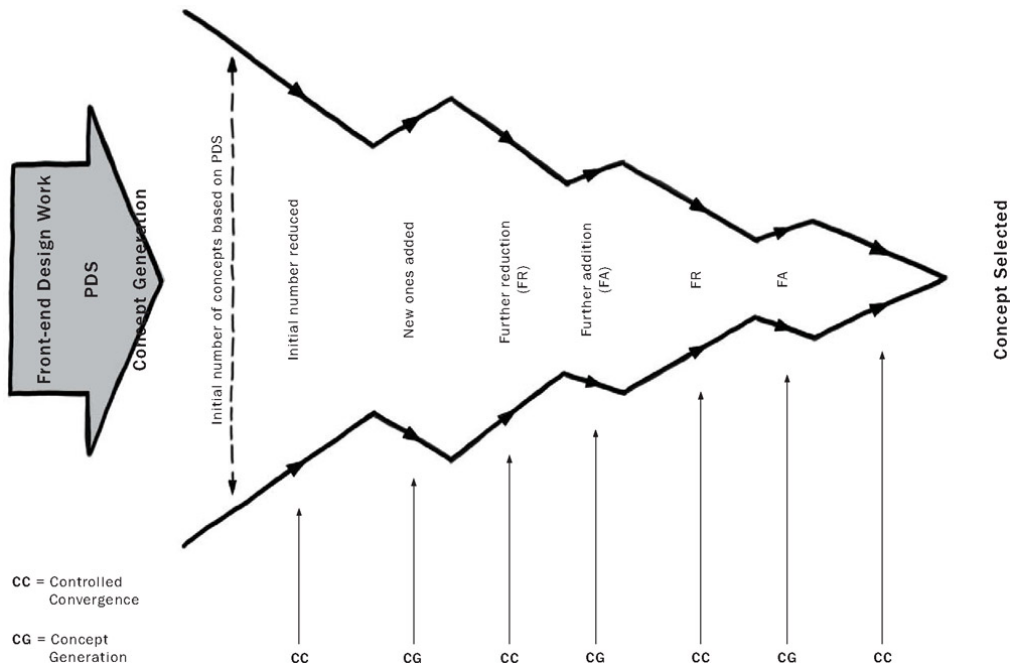


Figure 1.1: The design funnel with divergent and convergent phases [2]

1.2 Requirements Engineering

As an input to the funnel presented in [Figure 1.1](#), we look to other tools for inspiration on functional requirements for the simulator and its user interface. To initiate the process, we search for a number of diverse tools, that include a visual simulator. We look for diversity in the tools, meaning that we look for tools from different modeling domains e.g. Timed Automata, Petri Nets, SysML. This diversity might lead to ideas and requirements that are new to simulators for our domain, TIOA. We inspect some of these tools and collect a list of features and question users about which of these features they find important. Finally, we evaluate on the users' responses and gather a final list of functional requirements for the design process. Note that we actually collect two different lists of features, one for the functional features of the simulator and another for the User Interface (UI). The latter is used for inspiration for the UI and not as an input for the funnel, since that may limit the creativity of our sketches. Some UI examples of features are the ability to zoom and use grayed out elements for inactive actions. These UI features are not presented to any users.

Tools The research for tools has led us to a 16 different tools. Inspecting each of them is an extensive task, so we instead choose six tools to inspect. When choosing tools we consider the popularity and maturity, both as a research and enterprise tool. Another consideration is the domain diversity, to avoid ending up with six different tools from the same domain. With these considerations in mind, we present the final tools for inspection in [Table 1.1](#).

Tool	Domain
UPPAAL	Timed automata
Sparx Systems Enterprise Architect	SysML: activity diagrams, sequence diagrams, state machines and more
IAR Visual State	UML state machines
CPN Tools	Colored petri nets
PRISM	Discrete- and continuous-time Markov chains, Markov decision processes, probabilistic automata, probabilistic timed automata
TAPAAL	Timed-arc petri nets

Table 1.1: Tools used for inspection and their modelling domain

Inspection The inspection of the tools is performed independently by each member of this group, resulting in two separate lists of requirements. Doing the inspections

independently, we expect to find a greater range of requirements, as we may look for different features in the tools.

We do not intend to make a comprehensive list of all features in each of these tools, but rather get a general sense of the range of features. Making a comprehensive list would require us to be experts in each tool, and spend a lot of time inspecting them. Instead, we time-box the inspections to 30 minutes each. During the inspections, we get a first-hand experience with the tools, but also allow second-hand experiences through Internet research or videos. The second-hand experience is useful for discovering things we might have missed since we are newcomers to most of the tools.

After each inspection, we discuss our findings and make a joint list of requirements. The list of findings for functional and UI features can be found in [Appendix A](#). The features are sorted by the number of occurrences across the different tools.

Questionnaire To get an insight into what a user would like from a simulator, we prepare a simple questionnaire based on the discovered features. For each feature the user is asked to rank it as either *important* or *not important*. The questionnaire can be seen in [Appendix B](#). The group of users we question, includes students, professors, Ph.D. students, etc. from the Department of Computer Science at Aalborg University. The experiences of the group range from novice to expert users. We require that the users must have some experience with modeling and simulators e.g. they have used a simulator as a part of their research.

Results In total, we have questioned 18 people. The questions for feature #4 and #16 were only answered by 17 people. Only 14 people answered the question for feature #19 unanswered. The reason for leaving the questions unanswered may be that the person did not understand the feature. The questions that were left unanswered have been omitted from the calculated percentages in [Figure 1.2](#).

The results of the questionnaire are listed in [Figure 1.2](#). The mean value of all answers (the average of people answering *important*) is 60 %. We primarily consider the features above the mean value as candidate features i.e. features we will consider for the simulator.

[Figure 1.2](#) shows a clear need for the features #1, #2, #3, and #21 as all the users we asked have picked them as *important*.

In [Table 1.2](#) we present a prioritized list of features that should be considered in the design of the simulator. The list and prioritization are primarily based on the percentage of people who marked a feature as *important* and secondly on which candidate features we believe are interesting features. The prioritization is done according to the MoSCoW method. Of the 16 candidate features above the mean value, we choose the 12 shown in [Table 1.2](#), as features we will further explore. We deem the features in the *must have* category as a Minimum Viable Product (MVP) for a visual simulator i.e. these are the features that should be available in a visual simulator (hence the *must have* category)[6]. The MVP features should be implemented to have a worthwhile usability evaluation.

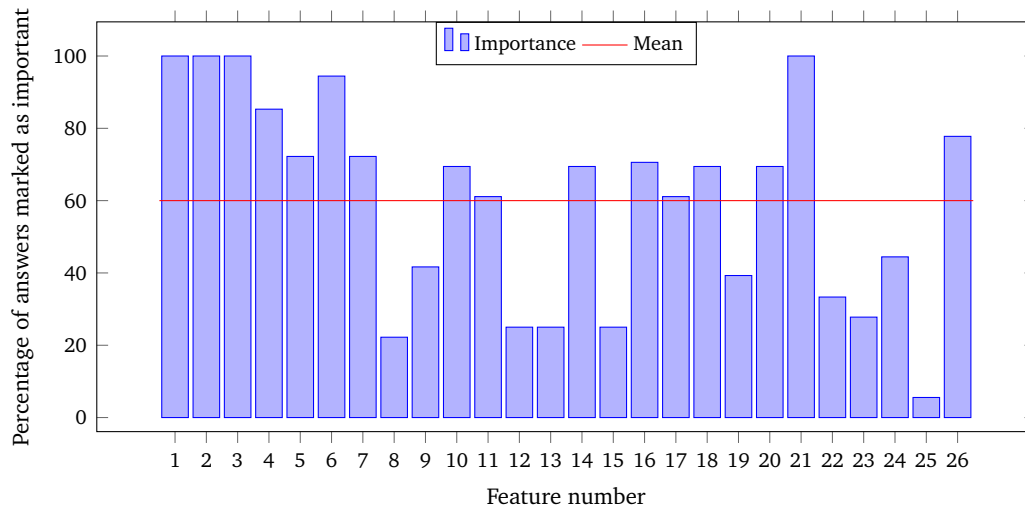


Figure 1.2: Percentage of users that marked a feature as important

Must have:

- #1 Manual stepwise execution of simulation
- #2 Overview of steps in a trace
- #3 Inspection of state values
- #4 Go back and fourth in trace
- #21 See traces from verifier in the simulator
- #6 Go to initial state

Should have:

- #26 Choose values to watch
- #16 + #10 Stop criteria / Simulation breakpoints
- #20 Keyboard shortcuts in simulator

Nice to have:

- #5 Random execution of simulation
- #7 Import / export trace

Table 1.2: Prioritized list of features (MoSCoW)

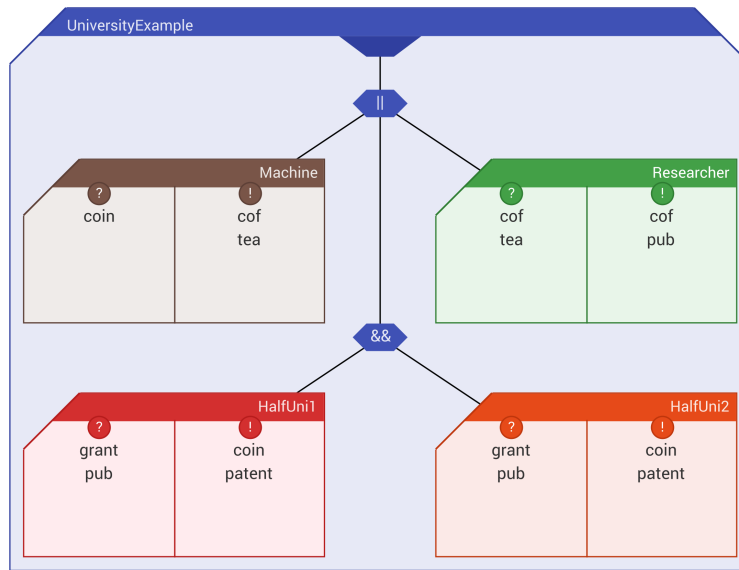


Figure 1.4: Example of a system view in Ecdar 2.0

1.3.2 H-Uppaal

This section is heavily based on Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1] and has been included here, with a few changes, to give a complete introduction of ECDAR 2.0 and what it is based upon. A precondition for ECDAR 2.0, is that it is built on the codebase of H-UPPAAL¹. H-UPPAAL is a different take on the UPPAAL Graphical User Interface (GUI). It improves on some issues related to creating large models in UPPAAL, for example by dividing the automata model into hierarchical components [4, 5].

The UPPAAL issues listed in Mouritzsen and Jensen [4] are mostly related to the complexity and readability of large models. The following are examples of issues they mention: Properties (e.g. invariants and clock resets) can be graphically placed away from their location or edge, locations do not have to be named (they are anonymous), and information can be hidden by overlapping objects. To combat these issues, they created H-UPPAAL, an IDE with continuous syntax checking, static code analysis, and background queries.

The representation of hierarchical timed automata in H-UPPAAL is presented in Figures 1.5 and 1.6. Figure 1.5 shows a hierarchical component, which contains sub-components for Customer, Waitress, and Kitchen. Figure 1.6 shows the Customer component.

1.3.3 Search for Related Research

As a part of our research process, we have investigated whether there is existing research on visual simulators for model checkers. However, we could not find any work that is

¹ECDAR 2.0 is forked from this commit of H-UPPAAL: <https://github.com/ulriknyman/H-Uppaal/tr..>

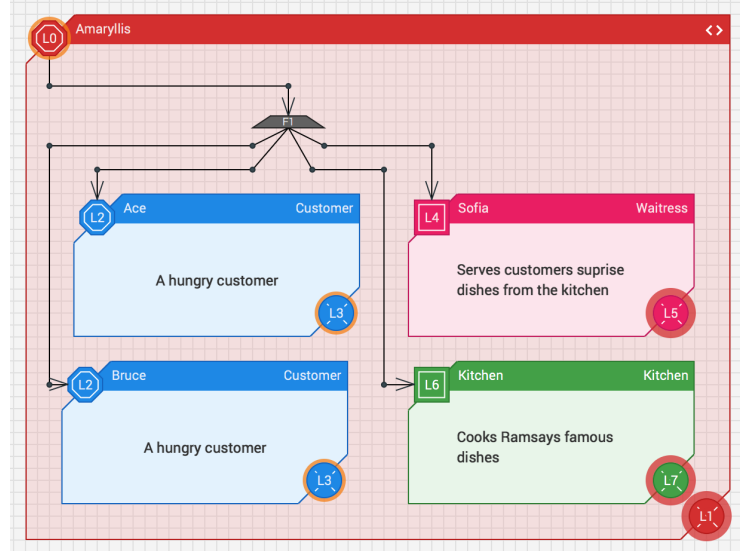


Figure 1.5: Screenshot of a hierarchical component in H-Uppaal [4]

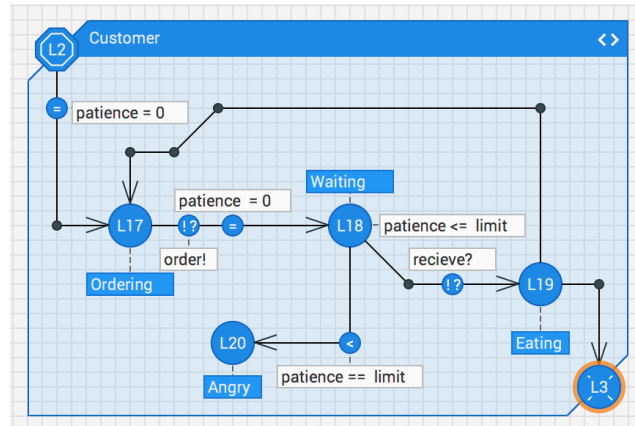


Figure 1.6: Screenshot of a component in H-Uppaal [4]

relevant for this project. This section describes our research process and effort.

We started the research process by looking for material specifically about visual simulators used for model checkers, but as we could not find anything this specific, we generalized the topic, so we looked for visual simulators (e.g. for UML or Markov chains), model checkers, and anything related.

For the research we used different tools to search databases for academic papers, but also more general Internet searches. Examples of the tools are *Google Scholar*, *CiteSeerX*, and *Primo*. The last tool also searches for books available at the university libraries. For general Internet searches we used *DuckDuckGo* and *Google*. With those tools we even looked for informal sources like blogs and videos that could be relevant for our topic.

The keywords used in our search are related to the specific field of simulators and model

checking, but also covered model checking tools. Some of the relevant keywords used in our search include: "model checking", "simulator", "uppaal", "visualstate", "simulation testing", "model-based simulation", etc. Besides using keywords for search, we also looked at related papers by researchers behind some of the model checking tools.

A great effort was spent on searching for related work, but none of it yielded any results. Since we could not find any research on visual simulators for model checkers, we assume there is not much research in this area. Most research in the field of model checking might be spent on the engines and formalism, not the user interface and the visual simulator.

2 Design

In this chapter we will put the theory of Benyon [7] and Pugh [3] into practice and go through three phases of concept generation (sketching) and concept convergence (evaluation) of the *must have* features, in order to decide on the final design of the simulator of ECDAR 2.0.

2.1 First Phase

As preparation for this first phase of the design funnel, we settled on a number of functional requirements (see [Section 1.2](#)), and prioritized them using the MoSCoW method. In this phase, we will mainly consider the features included in the *must have* category, as the questionnaire responses deemed them as the most important i.e. they are the features of the MVP. Each feature has mainly been sketched by itself i.e. without the other *must have* features on the same sketch.

2.1.1 Sketches

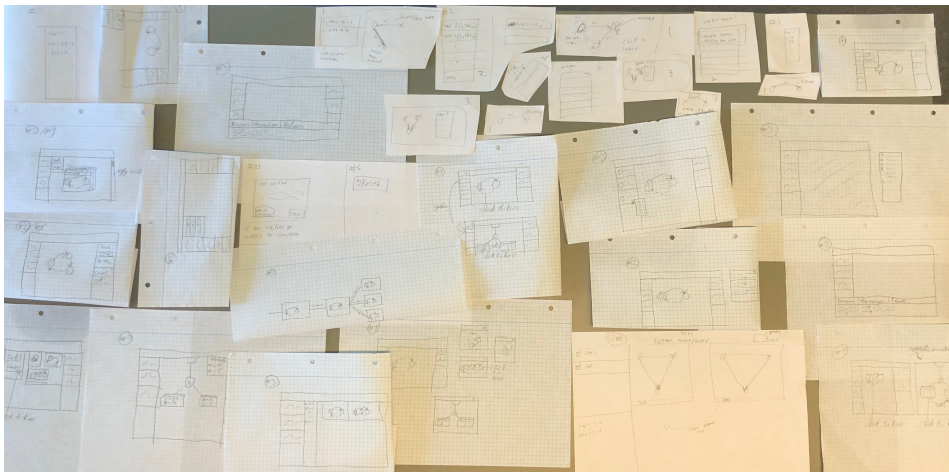
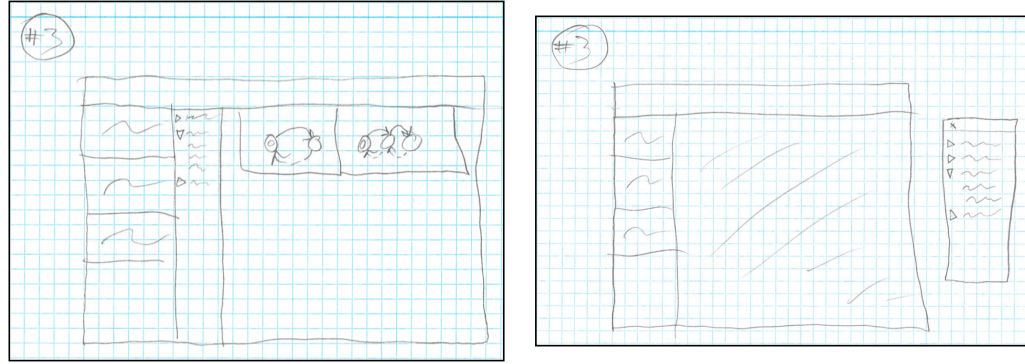


Figure 2.1: Overview of all the sketches generated for this phase

In this section, we highlight some of the sketches generated during the first phase. An overview of all the sketches generated in this phase can be seen in [Figure 2.1](#). The sketches included in this section are not necessarily the best concepts, but they rather represent the variety and level of detail expected in this phase. Each sketch has been assigned the id of the feature they represent e.g. [Figure 2.2a](#) represents feature #3 *Inspection of state values*.



(a) Inspection of state values as a list inside an integrated simulator

(b) Inspection of state values as a list in a separate window

Figure 2.2: Two representations of feature #3 *Inspection of state values*

Figure 2.2 shows two different representations of feature #3 *Inspection of state values*. Both Figure 2.2a and Figure 2.2b present state values (clock values and variables) as a list of component instances with values as a sublist for each instance. The major difference between these two sketches is the placement of the view for state values. In Figure 2.2a it is placed within the simulator window, which has the benefit of keeping related information in the same spot (an *all-in-one* view), but it also takes up screen estate that could be used for other simulation information e.g. the component instances. Figure 2.2b accommodates for the screen estate problem, by letting the user handle where the state values are placed since they are located in a separate window. This proposal provides the user with customizability, but it is possibly annoying to some users who prefer a single window. Both of the sketches have minimal details, making discussions about the sketches only concern the main concept and not the other details of the simulator.

Figure 2.3 presents two sketches of the feature #2 *Overview of steps in a trace*. The sketch in Figure 2.3a presents the steps in a trace as a simple list, where each step is a short text string describing the transition. Figure 2.3b is a more visual approach to the trace log. The steps are each represented by a snapshot of the component instances at the given step. The snapshots are placed in a stack, where the top element (the only snapshot that is fully visible) is the current state. The users can scroll (via mouse or scrollbar) through the snapshots, replacing the top element as they scroll. We will not go into the discussion of which proposal is best, as this example is only included to show the diversity in the sketches.

2.1.2 Categorization of Sketches

Sketching each of the *must have* features led to a large number of sketches (see Figure 2.1; each feature was sketched in multiple ways. Due to the many sketches, the problem arises of how we can select the *best ones*. There are different methods for resolving this

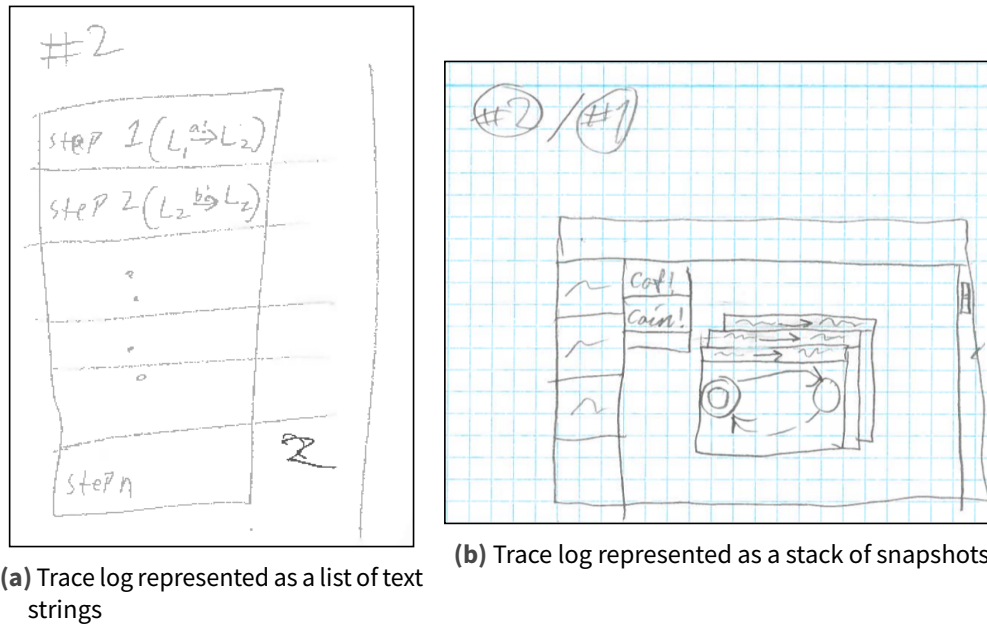


Figure 2.3: Two representations of feature #2 *Overview of steps in a trace*

problem, but while discussing the sketches we discovered an important concept, that should be settled before continuing with the next phase of the design funnel: Should the simulator be integrated or separated?

Integrated or separated simulator

The integrated simulator functions as a mode on top of the existing editor in ECDAR 2.0. Transitions are performed by clicking directly on edges of the components, and each component instance can be inspected individually as the user expects from the existing editor.

The separate simulator is on the other hand separated from the editor in ECDAR 2.0 and may be presented in a separate window or tab. The main purpose of the separate simulator is to give an overview of the simulation. Such an overview may contain all component instances, the enabled transitions, and system values. All of the information related to the simulation is presented in the same view.

Instead of converging by eliminating and choosing sketches, we group the sketches into four categories. This method is inspired by *vision scenarios* presented in Aaen [8]. The method employs a quadrant system (four quadrants along two axes) where each quadrant is used as a category (line of development). We consider the following axes:

Integrated simulator / Separate simulator The simulator can be integrated as a special mode in the existing editor, or it can give an overview of the simulation in a separate view (e.g. window or tab).

New view elements / Reuse of view elements Requires the creation of new view elements that may not be familiar to the user of ECDAR 2.0 or reuses view elements for new purposes.

These axes were chosen based on the tendencies we observed in the sketches: They work in an integrated or separate simulator, and they either reused existing views or required new ones. The idea with the groupings is that we can pick one or two of the groups to pursue in the upcoming phase. The next section describes how we choose between the groups on the *integrated simulator* / *separate simulator* axis.

2.1.3 SWOT

During the categorization process, a new concept emerged, namely a simulator combining features from both the integrated and separate simulator modes. To help us gain an overview of the pros and cons of each simulator proposal, we made a SWOT analysis for each of them. The SWOT analyses can be seen in [Figures 2.4 to 2.6](#).

<p>Strengths</p> <ul style="list-style-type: none"> • Overview of the complete system • All-in-one view • Easy to locate errors • Easy to see how transitions affect components 	<p>Weaknesses</p> <ul style="list-style-type: none"> • A lot of information • Breaks with current design
<p>Opportunities</p> <ul style="list-style-type: none"> • Clear code separation • One place to add new simulator functionality • Hide irrelevant information 	<p>Threats</p> <ul style="list-style-type: none"> • Completely new view to implement and design

Figure 2.4: SWOT analysis for a separated simulator

Separated Simulator The separated simulator mode ([Figure 2.4](#)) has the benefit of presenting all the relevant simulation information in a single view. This gives the users a clear overview and makes it easy for the users to locate errors and understand the

<p>Strengths</p> <ul style="list-style-type: none"> • High amount of reuse • Consistent with the current design • Focus on relevant information • Direct interaction with models 	<p>Weaknesses</p> <ul style="list-style-type: none"> • No overview of the system • Hard to see how the system is affected
<p>Opportunities</p> <ul style="list-style-type: none"> • Use system views for overview • See multiple components / systems at once (e.g. open components in separate windows) 	<p>Threats</p> <ul style="list-style-type: none"> • Harder to separate code • Challenging to extend with new mechanics • Potentially a lot of refactoring

Figure 2.5: SWOT analysis for an integrated simulator

communication between component instances. Having a single view for the simulator may lead to an abundance of information, and the view may not be consistent with the current design of ECDAR 2.0. One opportunity may be to hide irrelevant information, so the weakness of having a lot of information on the screen is reduced. The main threat lies in the design and development of a completely new view. The threat is not major, as we expect it is possible to make an appropriate design and implementation.

Integrated Simulator The integrated simulator mode (Figure 2.5) is focused on direct interaction at the level of a component instance. This could make it easy for users to understand individual components and concentrate on relevant information for a specific context. It also has high reuse of existing views, which makes it familiar to the user. There is no *all-in-one* view like with the separate simulator, so it may be hard for the users to get a complete understanding of how the component instances affect each other. To accommodate these weaknesses, one could use system views as a way of getting an overview of the system, or perhaps make it possible to open multiple components at the same time (e.g. open components in separate windows). The implementation of the integrated simulator is much more dependent on the existing ECDAR 2.0 code than the separate simulator. This may make it hard to separate simulator code from editor code

<p>Strengths</p> <ul style="list-style-type: none"> • Overview of complete system • All-in-one view • Easy to locate errors • Easy to see how transitions affect components • Some UI reuse • Consistent with the current design • Focus on relevant information • Direct interaction with models 	<p>Weaknesses</p> <ul style="list-style-type: none"> • Partially breaks with current design • Potentially a lot of information in the same view
<p>Opportunities</p> <ul style="list-style-type: none"> • One place to add new simulator functionality • Hide irrelevant information • Use system views for overview • See multiple components / systems at once 	<p>Threats</p> <ul style="list-style-type: none"> • Completely new view to implement and design • Harder to separate code • Potentially a lot of refactoring • Long development time

Figure 2.6: SWOT analysis for a combined simulator

and could require a lot of refactoring. It is also more challenging to add new features like activity diagrams, as there is no standard place to add general simulation functionality like the *all-in-one* view of the separate simulator.

Combined Simulator The combined simulator (Figure 2.6) is a proposal, that combines the two other proposals to achieve the strengths of overview and direct interaction at the component level. The combined simulator does not inherit all of the weaknesses from the other proposals since some of the inherited strengths make up for the weaknesses. The main change is the threat of long development time. It is a concern since we would have to design and implement both the integrated and separated simulator.

2.1.4 Evaluation

Based on the SWOT analyses, the combined simulator seems to be the best solution. It has numerous strengths and the weaknesses might be reduced, if we seek some of its opportunities e.g. "Hide irrelevant information" can accommodate for "Potentially a lot of information in the same view". Our main concern is the "Long development time" threat, as we would like to design and implement a fully functional simulator, and that might not be achievable within our time frame.

We would argue that by choosing one of the original proposals, integrated or separated simulator, we can finish the development within our time frame, and at a later point, one could implement the other proposal, resulting in the combined simulator.

We choose to work on a separated simulator due to a few reasons: While the integrated simulator has "high amount of reuse", it still requires a large refactoring task of existing code, which can slow down the rest of the development. The strengths "High amount of reuse" and "Consistent with the current design" from the integrated simulator are functionally not important, and can rather be seen as design challenges for the separated simulator that we can explore in the upcoming design phases. We prefer to explore usability related strengths like "Easy to see how transitions affect components" than the just mentioned design challenges. We still acknowledge the strengths and opportunities of the integrated simulator, but choose the separated simulator as the initial approach for the design.

Since we have chosen to work with a separated simulator, we can discard the sketches from the integrated simulator category. We take notes of interesting ideas before discarding any sketch. In the next phase of the design funnel, we concentrate on sketching features in the context of a separated simulator.

2.2 Second Phase

In this phase we generate new concepts based on the choice of designing a separate simulator. We individually sketch each feature, as to get a diverse set of concepts for each feature. Based on these concepts we will perform another round of convergence, where we choose some of the concepts as the output of this phase.

2.2.1 Sketches

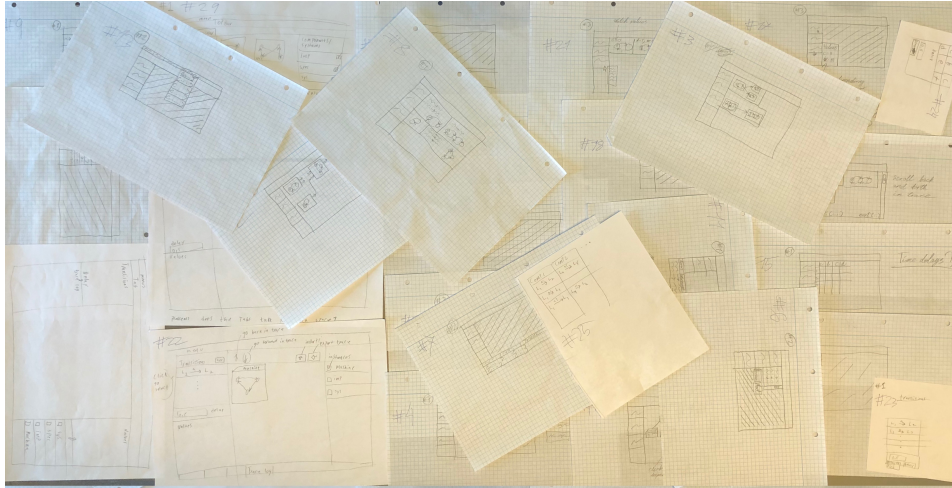
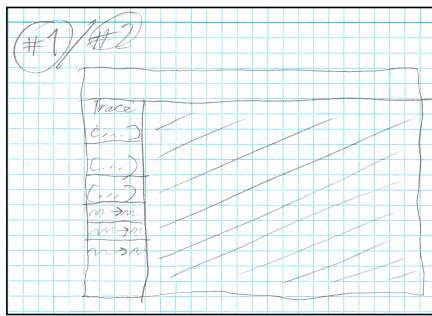


Figure 2.7: Overview of all the sketches generated for this phase

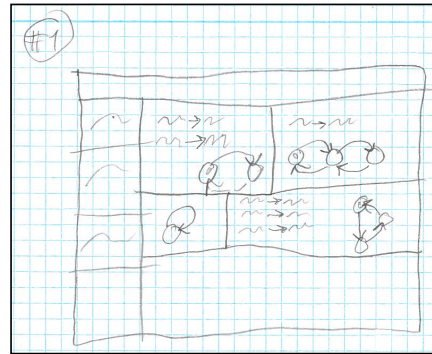
In this section we showcase selected sketches generated during the second phase. All of the sketches generated can be seen in [Figure 2.7](#). The sketches are again based on the *must have* features discussed in [Section 1.2](#). Note that we have not sketched *feature #6* and *feature #21*. The main difference between the first and second phase of sketching, is the choice of designing for a separate simulator. All of the sketches in this phase are designed in the context of such a simulator. Some ideas like showing values in a separate window ([Figure 2.2b](#)) may have been abandoned, since we have already discussed it, as a part of a group discussion, and decided on whether we should pursue it.

[Figure 2.8](#) presents two sketches of the *feature #1 Manual stepwise execution of simulation*. The sketch in [Figure 2.8a](#) is actually a combination of *feature #1* and *#2 (Overview of steps in a trace)*. The main concept behind this sketch is to reuse the left pane, previously used as file pane, to show the available transitions and trace log. The sketch also plays with the concept of a trace log and transitions: The trace is a timeline and the available transitions are future steps in this timeline. Instead of having two separate lists, the sketch combines the trace and transition lists into a single list. The trace items are placed chronologically above the available transitions. Clicking on a transition adds the clicked item to the trace and updates the available transitions.

[Figure 2.8b](#) shows an alternative to having a long list of available transitions. It shows



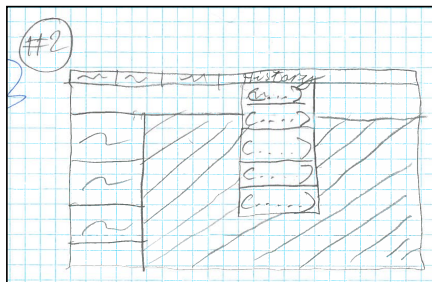
(a) Transition chooser and trace log combined into a single list of text strings



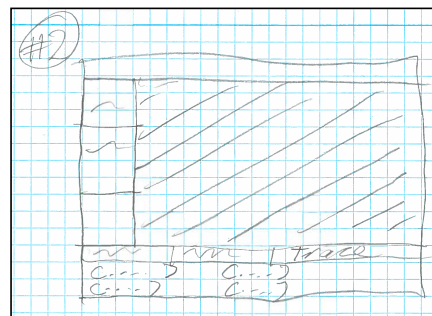
(b) Transition chooser on each component instance

Figure 2.8: Two representations of feature #1 *Manual stepwise execution of simulation*

which available transitions apply to a component directly on the component. This saves space as it does not require a separate view element to contain all transitions, and it should be easy for the user to understand which components are involved in a transition.



(a) Trace log as a menu item



(b) Trace log as a pane in the message container

Figure 2.9: Two representations of the feature #2 *Overview of steps in a trace*

The sketches highlighted in [Figure 2.9](#) are both examples of presenting the trace log (#2 *Overview of steps in a trace*) using existing views. [Figure 2.9a](#) imagines the trace log as a menu item. Clicking the menu item will show a drop down menu with each step of the trace as an item. It works like many people would expect from a *History* menu item in an Internet browser. One could even consider renaming it from *Trace* to *History* as that may be more recognizable to some users. It has the benefit of being easy to hide, so it does not clutter the simulator, but if the trace log is very important to the user's workflow, this benefit may be more of an annoyance.

The trace log presented in [Figure 2.9b](#) uses the existing message container of ECDAR 2.0. This container has tabs for *backend errors*, *errors*, and *warnings*. Clicking one of these tabs presents the users with a list warnings or errors (if any warning/error is present).

Using the message container for the trace log, also has the benefit of not cluttering the main simulator and being easy to hide. A concern with this sketch, is that it misuses the purpose of the message container, as the trace log is not really an informative message that the system reports to the user, but a simple description of the simulation states. [Figure 2.10](#) shows a reference image of the current message container in ECDAR 2.0. The warning tab contains a warning message with a link to the involved location.

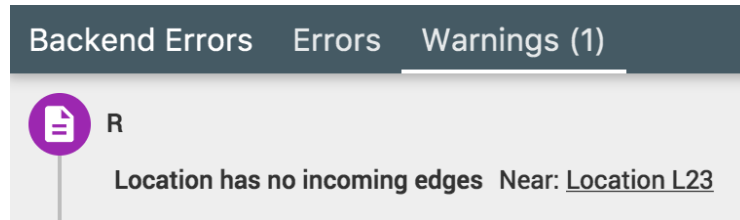


Figure 2.10: The message container in Ecdar 2.0

The sketches presented above, along with all of the other generated sketches, have been evaluated and discussed using the method described in [Section 2.2.2](#).

2.2.2 Ranking Sketches

As with the previous phase of sketching, we have created numerous sketches for each *must have* feature. The challenge is once again to pick some of the sketches as the outcome for the phase. For this phase we choose to rank each sketch according to a set of design criteria. Benyon [7] has collected a set of design principles, based on the work of different researchers. These principles are partially based on the work of Nielsen [9], which also coincides with the design principles of Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1]. Of the 12 principles proposed in [7] we choose to only evaluate our sketches based on four of the principles. If we were to evaluate using all principles, we expect the evaluation process to be very long, and some of the principles can be hard to apply to a single feature (easier to do for a complete system). The 4 chosen design principles are *affordance*, *consistency*, *feedback*, and *flexibility*.

Affordance "Design things so it is clear what they are for; for example, make buttons look like buttons so people will press them." [7]. We deem the affordance principle as important, because we want to design a simulator that the user knows immediately and intuitively how to interact with.

Consistency "Be consistent in the use of design features and be consistent with similar systems and standard ways of working (...) Both conceptual and physical consistency are important." [7]. The consistency principle follows the principles of [1]. We want the simulator to be consistent with the existing design of ECDAR 2.0, and may prefer solutions that reuses or looks like views from the editor of ECDAR 2.0.

Feedback "Rapidly feed back information from the system to people so that they know what effect their actions have had. Constant and consistent feedback will enhance the feeling of control." [7]. This principle is also related to the design principles of ECDAR 2.0 [1]. It should always be quick and easy to understand the current state of a simulation, what effect a transition has on the system, and what the previous steps of the simulation are.

Flexibility "Allow multiple ways of doing things so as to accommodate people with different levels of experience and interest in the system. Provide people with the opportunity to change the way things look or behave so that they can personalize the system." [7]. The design should allow for some customization like hiding irrelevant information. It should also provide the users with different ways of doing things, like having an action bound to a keyboard shortcut and also have the action available in the toolbar.

We rank each sketch on a scale from 1 to 5 for each of the principles. The score for a sketch can be tallied as the aggregate score for its principles. We do however give each principle a weight that the principle score should be multiplied with, as we believe some principles are more important in our design, and should therefore have a greater influence on the aggregate score. The weight for each principle: *affordance* (x2), *consistency* (x1.5), *feedback* (x1), and *flexibility* (x0.5). The weights are based on how important we believe each principle is. Table 2.1 shows an extract of the rankings table presented in Appendix C. This extract shows how each sketch is assigned an identifier (*sketch no.* column), which feature it represents (*feature* column), and what score it has received for each design principle.

		x 2	x 1.5	x 1	x 0.5	
Sketch no.	Feature	Affordance	Consistency	Feedback	Flexibility	Total
#1	#1	4	3	4	1	17
#2	#1	1	2	5	1	10.5
#3	#1	2	2	4	1	11.5
						⋮

Table 2.1: Extract of the sketch ranking table. The full table can be found in Appendix C

As with the sketching process, we each individually fill out a ranking table, and then discuss the scores of each sketch, until we agree on a final score. This process is inspired by the Planning poker estimation technique, where each individual estimates tasks by themselves and then discuss the task, before ending on a joint estimation [10]. The rankings presented in Appendix C are the joint rankings for all sketches drawn in this phase.

Partitioning

Our goal with this phase is not to just pick the highest scoring sketch, but to get inspiration for the design and pick a few sketches that can be used as the base design for the next phase. We therefore want to ensure that no sketch is discarded before we have discussed its pros and cons. To facilitate the discussion and converge on a few sketches, we partition the rankings table ([Appendix C](#)) into three partitions: The low scoring partition (score ≤ 12), the middle partition (score in interval of $12 < \text{and} \leq 16.5$), and the top scoring partition (score ≥ 17).

The sketches in the low scoring partition generally do not follow our design principles, or might focus on a single principle with low weight. We discuss each sketch in this partition quickly, with a focus on what the principle does well, before discarding the sketch.

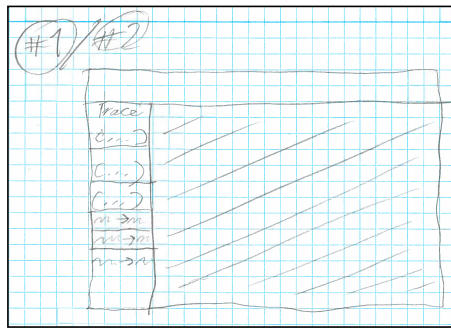
The middle partition mainly contains sketches that have a good *affordance*, but score low on the other principles. We also discuss these sketches before discarding them, but we might also consider merging the sketches with the sketches of the top scoring partition.

We consider the sketches in the top scoring partition as potential base designs, that should be build upon in the third phase. These sketches have an overall high score, but might score low on the least prioritized principle, *flexibility*. Note that we intended to only draw a single feature at a time, but some of these high scoring sketches actually represent multiple features. We allow this, since it can sometimes be hard to represent a concept without the context of the other features.

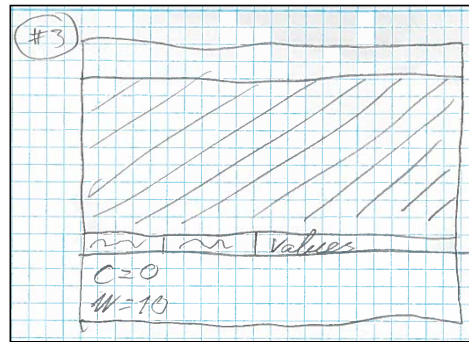
2.2.3 Results

For each sketched feature we choose one of the sketches in the top scoring partition as the base design. In the cases where there are multiple top scoring sketches for a feature, we discuss and choose the design we prefer. The sketches that represent multiple features, we either choose to be the base design for all of the features it represents or, if possible, select a specific feature from the sketch. Below we describe each of the sketches that we have chosen as base design.

Sketch #5 [Figure 2.11a](#) presents *sketch #5*, which is among the top scoring sketches. This sketch is also described in [Section 2.2.1](#). We have chosen this sketch to represent a base design for features *#1 Manual stepwise execution of simulation* and *#2 Overview of steps in a trace*. The trace log and transition chooser are placed in the left pane, staying consistent with the existing design, and the users can click on an available transition to execute it. To support the feedback principle, clicking on a transition immediately adds a new element to the trace log and updates the available transitions. The sketch does not offer much support for the flexibility principle, but we can easily imagine extensions to the design that for example add keyboard shortcuts to select and execute transitions. We were not satisfied with any sketches of feature *#4 Go back and forth in trace*, but believe that the feature can be designed in a refined version of this sketch.



(a) Transition chooser and trace log represented as a combined list



(b) Values as a pane in the message container

Figure 2.11: Sketch #16 and sketch #26

Sketch #16 Figure 2.11b presents sketch #16. It is also among the top scoring sketches. It represents the base design for feature #3 *Inspection of state values*. This sketch places the state values in a separate tab of the message container, that is previously used to show error/warning messages. Placing the state values in this container is consistent with the existing design, and the affordance is also deemed to be great, as the values are static text (no need to click them) and we expect that message container already has great affordance. The message container is also effective for feedback, since it can automatically pop-up when values are updated, or if the user keeps the container open, it can simply just update the text to show the effect of an action. We do not think the design is very flexible, but it can at least be resized to match the user's needs.

Sketch #26 Figure 2.12 presents sketch #26. This sketch is one of the top scoring sketches that represent multiple features (features #1, #2, #3). We choose this sketch to represent a general base design for the simulator. The sketch reuses the left and right pane of the application, to show trace log, transition chooser, values and shown/hidden processes. This should be consistent with the existing file and query panes, and the affordance is also expected to be good. It should be easy to get an overview of the system status, but it does not add anything special to the feedback principle. Lastly the sketch is very flexible in that it should be customizable, so the user can resize the elements in the panes and change their ordering. Note that the sketch also represents the features from the two previously mentioned sketches, and that might create a conflict in the design e.g. should values be in the message container or in the right pane. We aim to solve these conflicts in the third phase of the design. At the center of sketch #26 are the processes and systems involved in the simulation. Each process (instance of a component) is represented by its corresponding component, which for example can be used to highlight the current the state of the system.

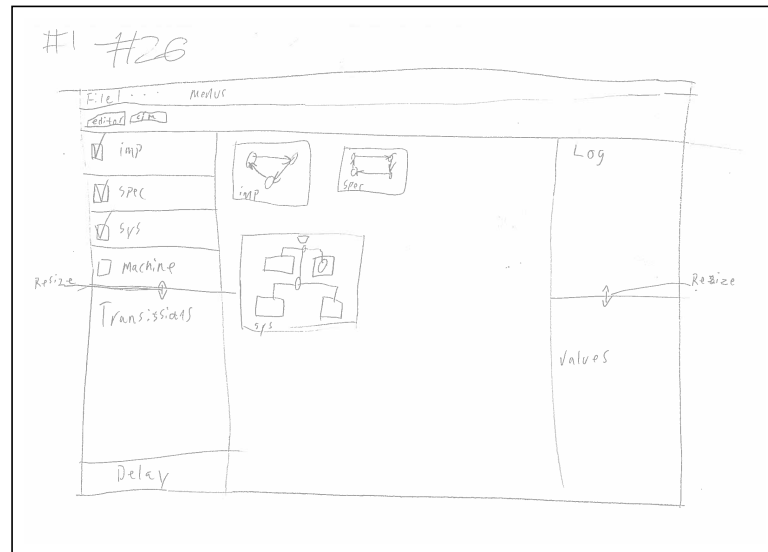


Figure 2.12: Sketch of the simulator with features designed as pane elements

2.3 Third Phase

The third and final phase of our design process also involves further concept generation and convergence. We choose this phase to be the last, as it results in a full design of the simulator and the important features. We could have introduced more design phases, which would help us design details of the simulator, but we deem the result of this phase adequate for implementation. The result covers all of the *must have* features, and we would like to implement the resulting design from this phase, before going in-depth with the details of the design, as the implementation might change some of the design.

2.3.1 Sketches

This section describes some selected sketches from the third phase. An overview of all the sketches generated in this phase can be seen in [Figure 2.13](#). These sketches combine the base designs, from the second phase, to present a simulator with all of the *must have* features. The sketches generated in this phase are not stuck with the base designs from the previous phase, we still allow for new concepts. As with the other phases, the sketching process is an individual exercise, so each of us sketch our ideas for a full simulator.

Common to the sketches in this phase is the base design from *sketch #29* of [Section 2.2.3](#). The sketch describes a general design that reuses the left and right panes of ECDAR 2.0. Each of the sketches may use the panes differently.

[Figure 2.14](#) presents a proposal for the full simulator. It uses the base design of *sketch #5* ([Section 2.2.3](#)) for the trace log and transition chooser. The values are located in the message container, as proposed by *sketch #16* ([Section 2.2.3](#)). The sketch also shows

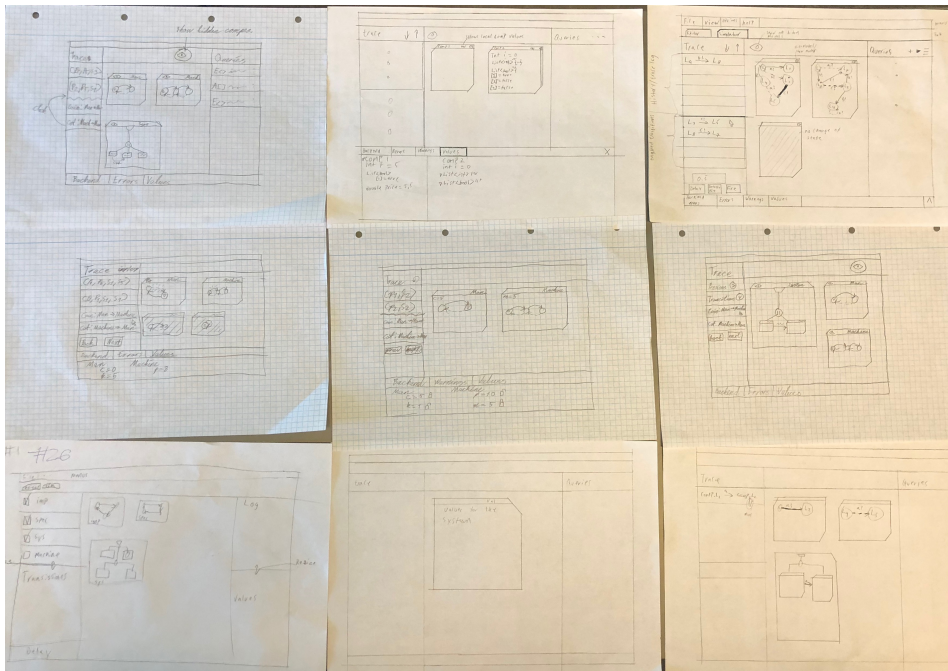


Figure 2.13: Overview of all the sketches generated for this phase

how the grey-out functionality works: When users hover on a trace log element, the middle part of the simulator highlights the processes involved in the transition, and greys-out the elements that are not used. It has the purpose of making it easy for the users to understand the processes involved in a transition.

The sketch also includes a functional feature that we have not discussed previously. The feature allows users to choose which processes should be visible in the simulator. It makes the design more flexible, since it allows the users to customize the main view. In the sketch it is visible as an "eye" icon in one of the top bars. The feature is related to the *hide irrelevant information* opportunity of the SWOT analysis for the integrated simulator [Figure 2.4](#). Please note that the feature is also included in the other sketches of this section.

The sketch also includes the query pane in the simulator. The query pane has previously only been available in the editor, but to create some consistency between the two modes, editor and simulator, we also want it to be available from the simulator. Queries are also very related to the simulator, as the users might want to see the resulting strategy of a query in the simulator.

[Figure 2.15](#) is a less detailed sketch, that changes the design of the trace log and transition chooser. Instead of having them as a single list, representing a continuous timeline, it proposes a design where they each are "sub menu"-like elements, that can be collapsed and expanded. If the users do not want to see the long trace, they can collapse that menu (the "previous" element in the sketch) and still be able to see the available transitions. This also allows for a more flexible design. The transition chooser and the

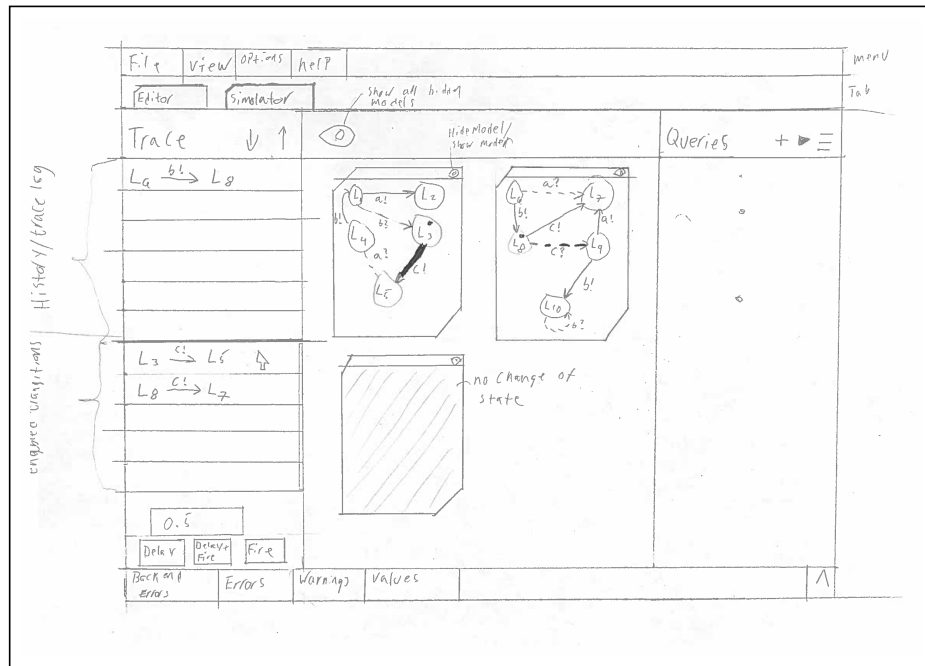


Figure 2.14: Sketch #1 of the full simulator

trace log can be placed above each other to imitate the timeline concept.

Figure 2.16 presents sketch #3 of the full simulator. This sketch focuses on how the values are represented. As proposed in the base design from Section 2.2.3, the state values are placed in the message container. The message container tab for values is separated into segments for each process. Each segment contains the variables and clocks for a given process. The sketch also proposes a secondary way for viewing values, that is consistent with the way that users must edit variables on a component. The users can press a button on a process to see an overlay with the process' values. Having two ways of presenting values adds to the flexibility of the design.

2.3.2 Simulator Configurations

The number of sketches generated in this phase was lower than the previous phases, as the sketches are more detailed and include multiple of the discussed features. All of sketches from this phase are presented in Figure 2.13. Having few sketches enables us to spent time on discussing each sketch and specific details. From these discussions we note the things we like and propose three configurations for a simulator. The three configurations are very similar and include the same features, but the features might be represented in different ways. Table 2.2 shows how each of the configurations are built. The *common* column in the table, shows the features and representations each of the configurations have in common. The configurations have also been sketched (more detailed than previous sketches), as to make a common understanding of how the

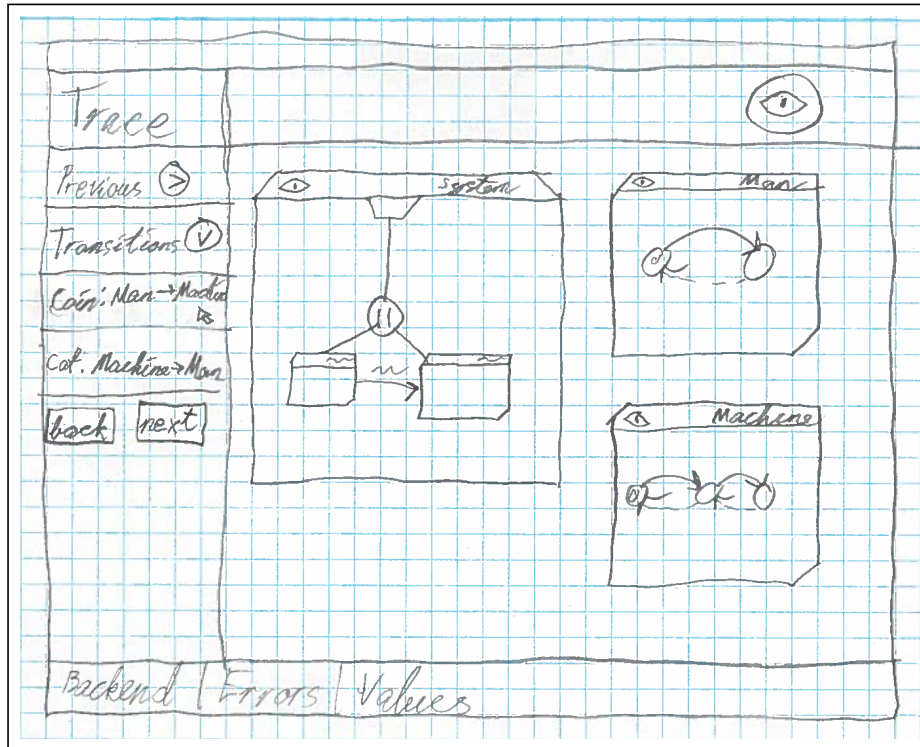


Figure 2.15: Sketch #2 of the full simulator

simulator configurations look. These sketches can be found in [Appendix D](#).

The three configurations of the simulator all satisfy the requirements of having the *must have* features proposed in [Section 1.2](#). Each configuration is based on the design of a separate simulator, a choice made during the first phase [Section 2.1](#). The design of the features have been evaluated to satisfy the design principles in [Section 2.2](#). Note that we do introduce new designs for features, that are inspired by the base designs of the second phase, and these new designs have not been evaluated based on the design principles. We believe the designs satisfy the principles and requirements well, since they are a result of other concepts that have been generated during the design phases, but have not formally evaluated them. In [Chapter 4](#) we present a usability evaluation of the simulator, that is used to evaluate the design choices we have made.

Since all the configurations satisfy our requirements, we choose the one that we prefer for the final concept in the design funnel process. The design is a great foundation for the upcoming activities e.g. it makes creating specific tasks for implementation easy. We still allow changes to the design, since we may discover issues or new concepts for it. We choose to proceed with *configuration 2*. A small version of the sketched configuration can be seen in [Figure 2.17](#).

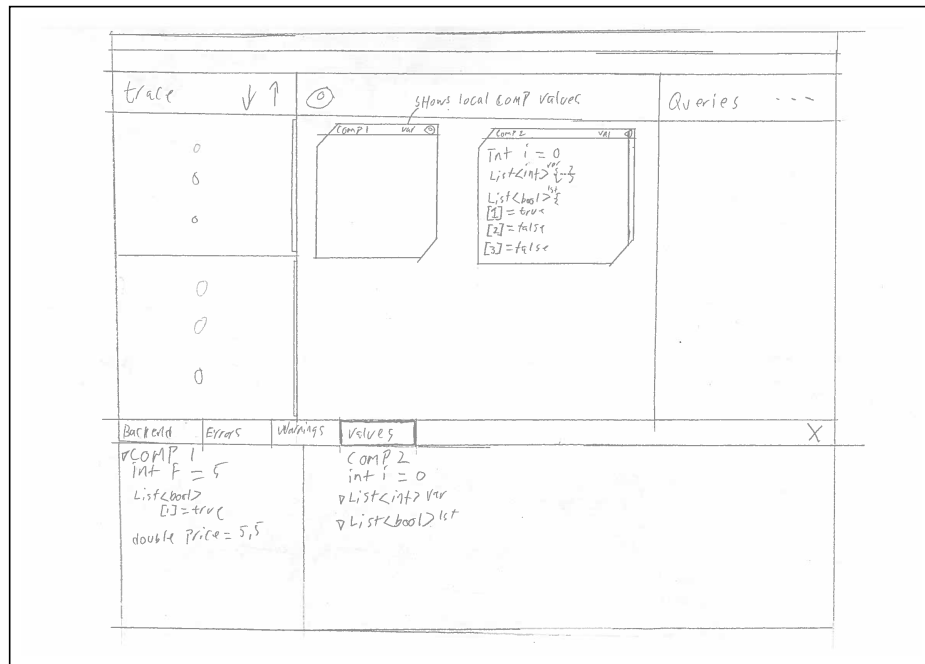


Figure 2.16: Sketch #3 of the full simulator

Common	Configuration 1	Configuration 2	Configuration 3
Buttons for executing transition	Values on process instances	Values in message container	Values in right pane
Highlight and grey-out processes when choosing trace/transition	Combined trace and transition pane	Trace pane with sub menus	Combined trace and transition pane
Reset at the top of the trace pane	Includes query pane	Includes query pane	No query pane in simulator
	Hide button on each process instance	Hide button on each process instance	Hide instances from right pane

Table 2.2: The different configurations of simulators

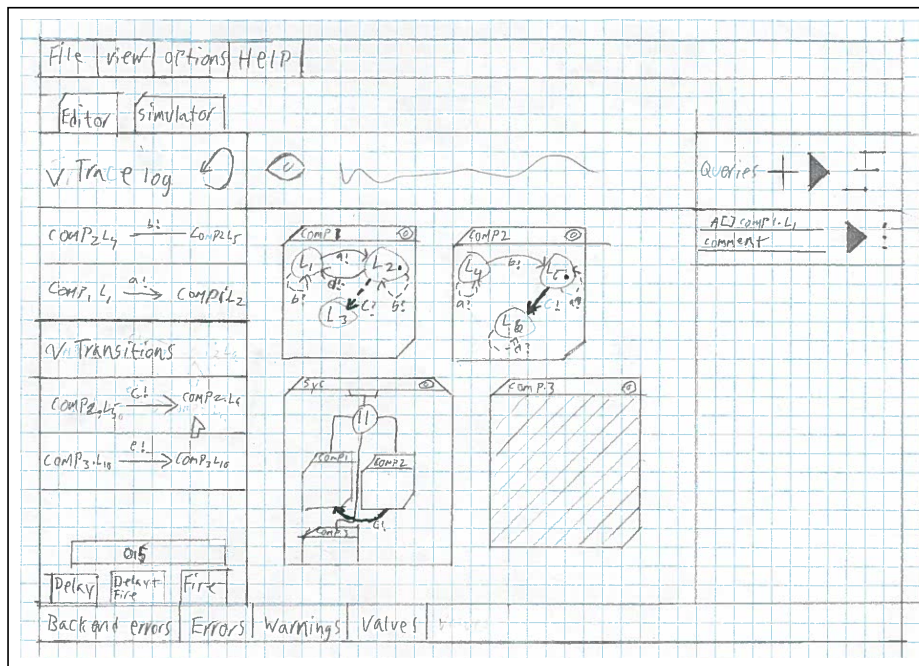


Figure 2.17: Configuration 2 of the simulator. A large version of the sketch can be found in [Appendix D](#)

3 Implementation

In this chapter, we go into details with regards to how the sketches have paved the way for the implementation of the visual simulator in ECDAR 2.0, and describe some of the challenges we had in regards to the backend.

First, we start with explaining how we managed to connect with the backend and make advances in a simulation, which also led to some changes in the architecture of the data access layer of ECDAR 2.0. Then we present what the implemented simulator looks like compared to the chosen configuration in [Section 2.3](#) and discuss the changes between them.

3.1 Reverse Engineering the Communication with the Backend

Going into this project we expected the backend to work without any major issues. It already works fine for the queries used in ECDAR 2.0 and the ECDAR 0.10 backend has functionality for simulation i.e. it is already used for simulations in UPPAAL TIGA. It turns out that it was not as easy to use as we expected. We have used the versions of the backend that both ECDAR 0.10 and UPPAAL TIGA use. UPPAAL TIGA contains the latest version of ECDAR. Using both backends we experienced crashes caused by segmentation faults.

The backend is essential to the work we make on the simulator, so fixing the problems we experienced were our top priority, and we needed to solve them before proceeding with other tasks.

We had no explicit documentation for either the ECDAR 0.10 or UPPAAL TIGA engine, so we used the available documentation for the UPPAAL engine to get an understanding of the other engines. There are a lot of commonalities between these engines, so we also expect the UPPAAL documentation to have something related to the libraries for the other engines.

The UPPAAL documentation is comprehensive, with guides for how to establish a connection with the backend, how to do verification, and advance in a simulation. Some of the methods available in the ECDAR 0.10 libraries were not available in the UPPAAL documentation or had different names. This led us to explore the library by a *trial and error* approach.

As the *trial and error* approach did not lead us to a solution, we turned our attention to the `socketserver`¹ in order to see how ECDAR 0.10 communicates with the backend. After some trials with looking at the messages send between ECDAR 0.10 and the backend, we requested and got access to the GUI code of ECDAR 0.10. With the GUI code we expected

¹Bundled together with the backend of ECDAR 0.10 and UPPAAL TIGA

to get a better understanding of how it communicates with the backend and specifically how it advances in a simulation. We also made the same inspections on the backend for UPPAAL TIGA.

Having inspected the GUI code for ECDAR 0.10 and the messages that both ECDAR 0.10 and UPPAAL TIGA send to the socketserver, we tried to imitate the same steps and method calls. This did however not work as intended as we ended up causing *ServerExceptions*. The confusing part of the exceptions was that they were not really consistent and showed different causes (i.e. messages) when we tried to handle the exceptions. Luckily we discovered, by pure coincidence, that calling the same method multiple times consistently ended up returning the correct results. So to advance in a simulation, we have to call the method, that takes a step in a simulation, multiple times.

The whole debugging and reverse engineering lasted for about two to three weeks, with the conclusion that when we try to take a step in a simulation then the method has to be executed multiple times.

3.2 Architecture

In Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1] we describe how the frontend of ECDAR 2.0 communicates with the ECDAR 0.10 backend, but with the introduction of the simulator, that architecture needs to be revisited in order to make the simulation functionality available.

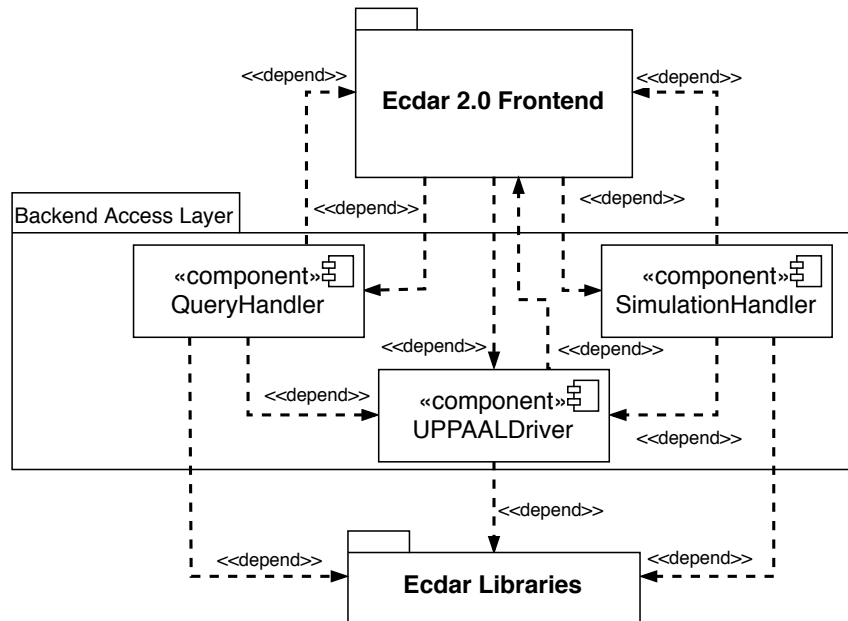


Figure 3.1: An UML 2.5 [11] component diagram of the architecture of the backend access layer of Ecdar 2.0

Figure 3.1 presents the updated architecture of the backend access layer for ECDAR 2.0.

The *EcdarDocument* component/class, found in the architecture of Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1], has been omitted from this figure due to simplicity, but note that the new architecture has not changed its relations. The biggest change in the architecture is the introduction of the two classes: *QueryHandler* and *SimulationHandler*. With the introduction of the two classes the *UPPAALDriver* has been refactored and its only responsibility is to keep track of engines (i.e. instances of the backend), by creating and releasing them.

In the previous version of ECDAR 2.0, the *UPPAALDriver* also had the responsibility for handling queries [1]. This has instead been moved to its own class, namely the *QueryHandler*, which now has the responsibility of sending queries to the backend, through the use of the *Ecdar Libraries*. In this way, if a component in the frontend needs to get a query checked, it should just make use of the *QueryHandler*, as it also communicates with the *UPPAALDriver* in order to establish the connection with the backend.

The newest addition to ECDAR 2.0 is the visual simulator, and in order to keep the new responsibilities and classes well-defined, in regards to the backend access layer, we introduce the *SimulationHandler*. The responsibility of the *SimulationHandler* is to make the simulation related functionality easily accessible to the frontend. This includes functionality to initialize the simulation, go back and forth in traces (*feature #4* and *feature #6*), take a step in the simulation (*feature #1* and *feature #2*), view a strategy from a query (*feature #21*), and making simulation values and clocks accessible (*feature #3*).

Looking at the *must have* features in Table 1.2, all of the simulation functionality is now provided through the *SimulationHandler* and the restructuring of the backend access layer. In the following sections, we will go into details of how the different *must have* features have been implemented. But first we present an overview of the implemented simulator.

3.3 Overview of the Simulator

In this section, we introduce the implemented simulator and also compare it to the selected base design, configuration 2 (seen in Figure 2.17), from Section 2.3.2.

Figure 3.2 shows what the implemented simulator looks like in ECDAR 2.0. The simulator can be accessed by clicking on the play icon in the bar on the left-hand side, and the editor can be accessed by clicking on the house/home icon.

There are some differences, some bigger than others, when comparing the selected configuration with the implementation. Some of the changes include the left-hand bar instead of tabs, the swap of the transition chooser and the trace log, the "eye" feature, and how values are presented. We will in this section describe each of these changes.

In the selected configuration, the editor and simulator are split into two tabs at the top of the application. To switch to the simulator the users just have to click the "simulator" tab. When it came to implementing the tabs, we did not feel like the tabs were the best solution. We expect that when clicking a tab, a new similar view is opened i.e. clicking a

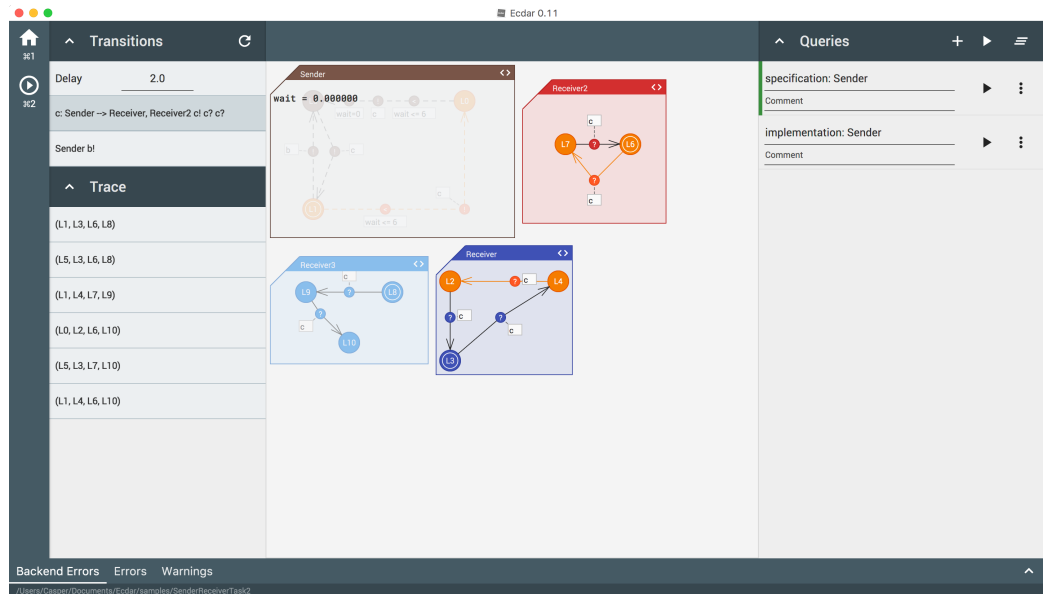


Figure 3.2: An overview of the simulator

tab in the editor we would expect a new editor window to be opened. The simulator is a completely different and separated mode, which is an idea the solution should afford. The tabs do however not afford this idea. Besides this idea, we could also not come up with a design for the tabs that is visually pleasing e.g. it did not look great if placed between the menu bar and the main view (editor or simulator), and the tabs should be separated from the main view. The solution we propose is the left-hand bar, where the user can click on icons or use keyboard shortcuts to switch between modes. The solution is inspired by other applications e.g. Slack and Visual Studio Code. They use this type of left navigation bar to switch between workspaces and main modes.

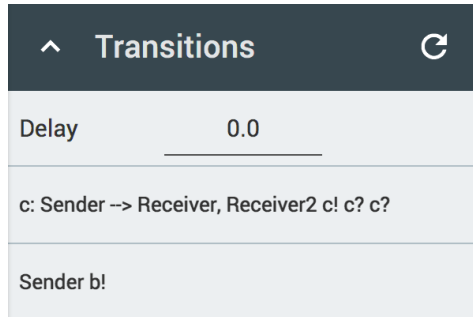
Another difference between the configuration and implementation is the order of the trace log and transition chooser. The original concept (presented in [Section 2.2.1](#)) was to represent both as a continuous timeline, but we realized that it would become impractical. The transition chooser and trace log are both placed in the same pane, and when one of them grows it would push the other view down i.e. whenever a new item is added to the trace log, the transition chooser would be pushed down. We saw this as a potential annoyance that could cause many misclicks.

A feature that is not implemented in the simulator, was the option to hide the irrelevant processes, also referenced as the "eye" feature. The feature does not support any of the *must have* features (presented in [Table 1.2](#)), so it has been postponed as we prioritize the features needed for a MVP. We advise presenting the "eye" feature to some users to see if it has any value to them.

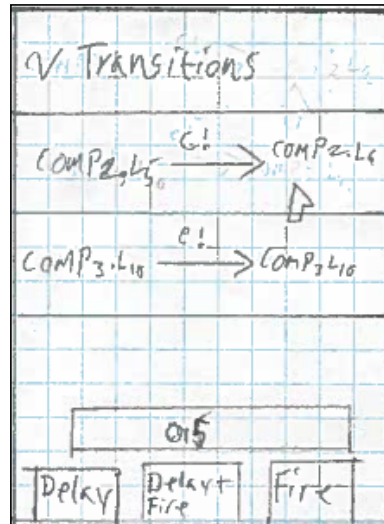
In the configuration, the state values are a part of the message container, but in the implementation, the values are shown as an overlay on the process. The argumentation and discussion for making this change can be found in [Section 3.6](#).

In the following sections we discuss and compare the individual *must have* features and their implementation.

3.4 Feature #1 – Manual Stepwise Execution of Simulation



(a) The implementation of the transition chooser



(b) The sketched version of the transition chooser

Figure 3.3: Implementation and the sketch of feature #1

Figure 3.3 presents the implemented transition chooser and the sketched version. The transition chooser is used to perform transitions, also called taking a step in a simulation. The users can add a delay to such a transition.

There are only slight visual differences between the implementation and sketched version. The main differences are related to how the users perform a transition and the reload button at the top of Figure 3.3a.

The transition buttons ("delay", "delay + fire", and "fire") in the sketch have been ditched in the implemented version as the same functionality is instead performed when the users click on a transition i.e. clicking a transition fires it, and clicking on a transition with a entered delays, fires the transition with the entered delay.

This behavior is also different from ECDAR 0.10, as it would require a double click where the first click is a selection, which highlights the involved edges and locations, and the next click fires the transition. In ECDAR 2.0 we find the first click unnecessary as ECDAR 2.0 uses mouse hover on transitions to highlight the involved edges/locations and grey-out the processes that are not affected.

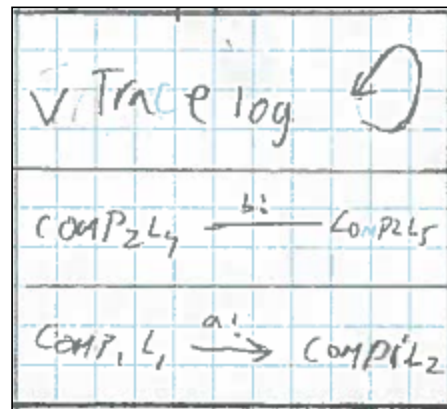
The reload button in the top-right corner of the implemented transition chooser, is used for reloading changes to the model and resetting the simulation. It is a necessary addition that will be discussed in Section 3.8.

As mentioned in [Section 2.2](#) one of our design principles is *flexibility*, which encapsulates the possibility for customization. We envision that the users should be able to place the pane elements (transition chooser, queries, and trace log) in any pane (the left or right panes) and in any order they would like. Every pane element has been implemented such that they work independent of any pane, which would make it programmatically easy to move the elements. Besides that the decoupling of views also contributes to higher code quality. Note that each pane element has a collapse/expand functionality to make the behavior of the elements consistent.

3.5 Feature #2 & #4 – Overview of Steps in a Trace & Go Back and Forth in Trace

^ Trace
(L1, L3, L6, L8)
(L5, L3, L6, L8)
(L1, L4, L7, L9)

(a) The implementation of the trace log



(b) The sketched version of the trace log

Figure 3.4: Implementation and the sketch of *feature #2* and *feature #4*

[Figure 3.4](#) presents the implementation and sketch of *feature #2*. This feature deals with how the trace log is represented and is related to *feature #4*, which concerns how the users can inspect the states of the trace. Both the implementation ([Figure 3.4a](#)) and sketch ([Figure 3.4b](#)) are designed to be visually and behaviorally consistent with the transition chooser (described in [Section 3.4](#)). The trace log pane element can be collapsed/expanded and hovering a trace log item highlights the state of the processes.

The implementation and sketch differs in how the text for a state is represented. In [Figure 3.4a](#) the text $(L1, L3, L6, L8)$ means that one process was in location $L1$, one was in location $L3$, and so on. The implementation shows the locations corresponding to the state. The sketch has more information like the synchronization, the state before and after the synchronization, and the name of the processes involved. We decided to only show the locations due to a few reasons:

1. The locations are assigned unique identifiers and we highlight the processes where they are located, so there is no need to also have the process name in the text.

2. We prefer for the trace log to only represent states. The synchronization in the sketch is not a state, but it shows the transition between states. The trace log elements could include information about transitions, but the elements themselves should only represent a state.
3. The sketch representation is also impractical because the text is more likely to become long, which could make it harder for the users to read.

Feature #4 is concerned with going back and forth in a trace. Clicking an item in the trace log, sets the simulation to the clicked previous state. It is consistent with how the users interact with the transition chooser. It is not possible to go forward in a trace, since stepping back removes items from the log, such that the current state is the latest.

The reload button in the sketch, also mentioned in [Section 3.4](#), was designed to just reset the current simulation i.e. it would go to the initial state and initialize all variables and clocks. [Section 3.8](#) discusses the functionality of the reload button.

3.6 Feature #3 – Inspection of State Values

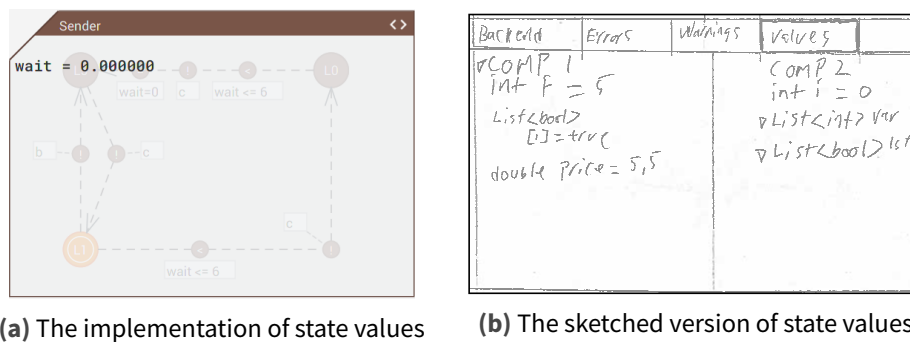


Figure 3.5: Implementation and the sketch of *feature #3*

The sketch and implementation of *feature #3* are presented in [Figure 3.5](#). This feature is concerned with where to place the state values of the processes. There is a great difference between the implementation and sketch. The implementation places the values on each process of the simulation. On a process the users can click the <> button (top-right corner of [Figure 3.5a](#)) to get an overlay with the values of that process. The sketch (see [Figure 3.5b](#)) proposes to place the values as a new tab in the message container, along the other tabs: *backend errors*, *errors*, and *warnings*. The users can click on the new *values* tab to see the state values for every process in the simulator.

We decided to not follow the sketched design due to the following reasons:

1. The message container is used across the whole application i.e. the users can access the message container from both the editor and simulator. The values are only related to a simulation and should not be accessible from the editor.

2. The message container is used for temporary messages about the system. The values, compared to the other tabs, are almost persistent messages, which do not belong in the message container.

Instead of using the sketched design, we looked at alternatives, that we had already sketched and used in the configurations of [Section 2.3.2](#). The two alternatives are presented in [Figure 3.6](#).

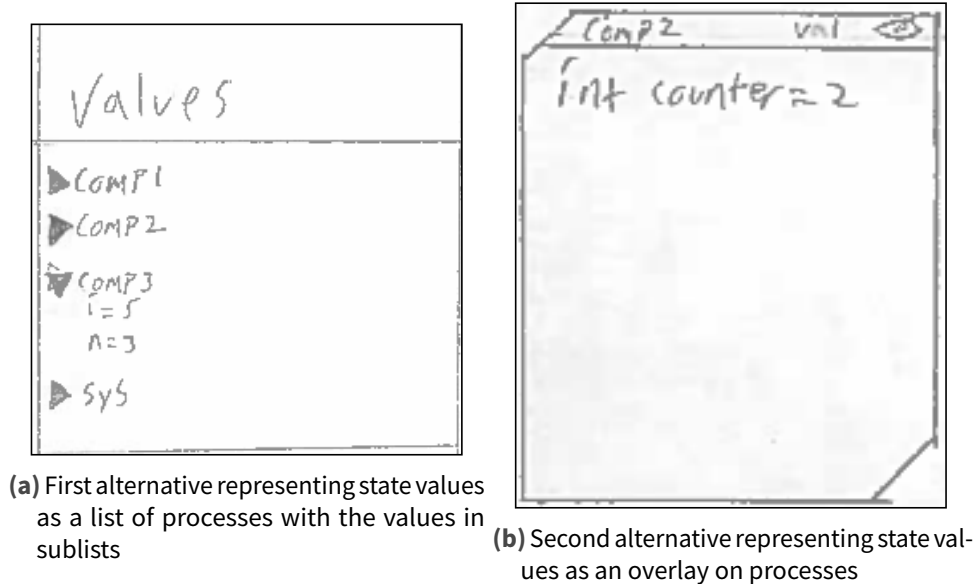


Figure 3.6: Two sketched alternatives to showing state values

The first alternative (seen in [Figure 3.6a](#)) shows a new pane element where each process can be collapsed or expanded to show its values. This solution is very similar to the solution found in ECDAR 0.10.

The other alternative (seen in [Figure 3.6b](#)) shows the values of a process, as an overlay on-top of a simulated process. The users can click the "val" button in the top-right corner to show this overlay.

As can be seen in [Figure 3.5a](#), we chose the second alternative. We like both sketches, but prefer this alternative because it is consistent with the location of the values in the editor.

3.7 Feature #21 – See Traces from Verifier in the Simulator

Feature #21 is concerned with how to get the resulting strategy from a query and presenting it in the simulator. In [Section 2.2.1](#) we state that this feature would not be sketched. It is hard to explicitly sketch the feature, since it uses the transition chooser, trace log, state values, and simulated processes. This section will instead present how to trigger the feature.

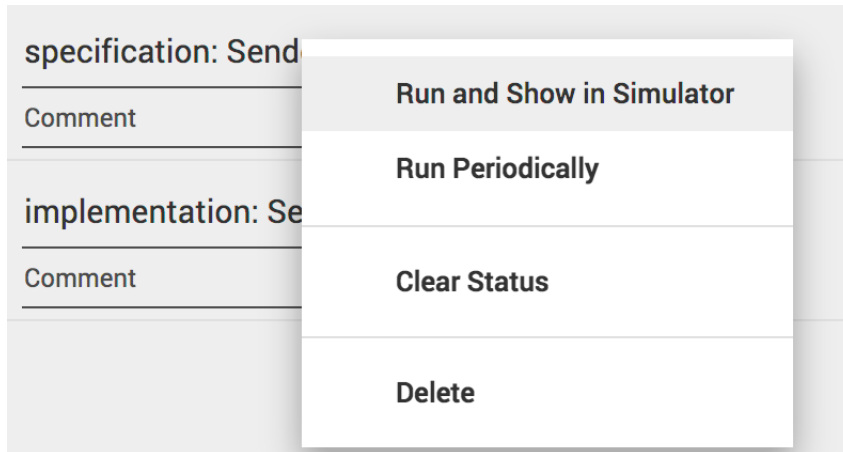
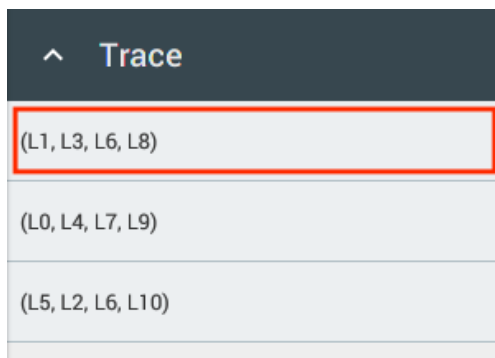


Figure 3.7: The implementation of the menu item "Run and show in simulator" found on a query

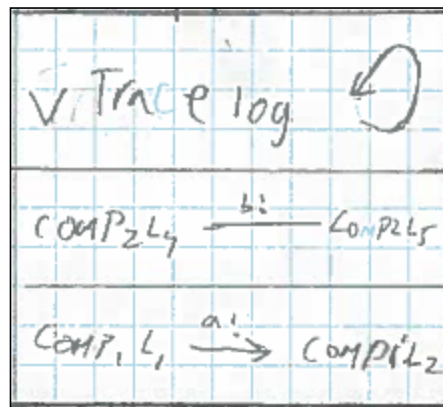
Each query in the query pane element has a *more* button (three dots), clicking it presents the users with a context menu. Figure 3.7 presents the context menu with the menu item "Run and Show in Simulator". Clicking this menu item will run the query, open the simulator, and update the trace log and transition chooser with the resulting strategy.

We also considered adding an extra button next to the *run* and *more* buttons, but that complicated the visuals, since there would be an extra button per query. We believe that adding the functionality as a menu item is a simple and satisfying alternative.

3.8 Feature #6 – Go to Initial State



(a) The implementation of the trace log with first item marked



(b) The sketched version of the trace log

Figure 3.8: Implementation and the sketch of feature #6

Feature #6 is the functionality to go back to the initial state. In [Section 2.2.1](#) we mention that we did not sketch this feature separately i.e. we did not make a bunch of sketches for the features, but it may have been included in later phases as a part of bigger sketches. The configurations presented in [Section 2.3.2](#) all include the feature as a reset button at the top of the trace pane. An example of this can be seen in [Figure 3.8](#). The reset button would, as the name indicates, reset the simulation, which corresponds to going to the initial state.

The implementation in [Figure 3.8a](#) does not include this button, as it has been repurposed for a reload functionality and placed at the top of the transition chooser. The reload button does still reset the simulation, but it also loads changes from the editor. In this way if the users add a new location to a component, they need to reload the simulator to see the new location on the given component.

It was necessary to add the reload functionality, because we wanted the simulator to preserve progress and not start the simulation over, every time the users enter the simulator.

The simulator may in some cases automatically reload, if there is no progress in a simulation i.e. if the users have not performed any transitions or are simulating a query strategy.

To go back to the initial state, the users have to click on the first element in the trace log (as marked in [Figure 3.8a](#)). This is also consistent with how we expect the users to interact with the trace log [Section 3.5](#).

3.9 Summary

In this chapter, we have looked at the challenges in regards to the communication with the backend and discovered that calling the same method multiple times consistently returns the correct results. The architecture has been revised to include the functionality needed to connect the frontend of the simulator to the backend.

The implemented simulator looks much like the chosen configuration (see [Figure 2.17](#)) with the main differences being: The tabs in the configuration have been replaced with a bar on the left-hand side, the transition chooser and trace log have been swapped, the "eye" icon together with the functionality to hide processes have been left out, and the design for state values has been changed to a design that is consistent with the editor.

4 Usability Evaluation

With the implementation of the *must have* features in place, we conduct a usability evaluation to validate the design choices of the ECDAR 2.0 simulator and check whether it has the features that users expect. We want to discover if the participants finish the tasks, if and where they get stuck, and if they find anything in the simulator confusing. We would also like to hear the participants about their input on how to further improve the simulator.

4.1 Participants, Setup, and Tasks

In total we have conducted the evaluation with nine participants. The participants are divided into two groups, namely novices and experts, where the experts have more than two years of experience of using a model checker and for the novices we require that they have used a model checker for at least one project. This leaves us with six novices and three experts in total. By having different levels of expertise in the group of participants we expect to identify different usability issues, as the experts might have different expectations than the novices.

When a participant finishes the last task, we carry out a debriefing interview (based on Benyon [7]), where each participant is asked about what was good about ECDAR 2.0, what was bad, and if they had any suggestions for changes or features. By having a debriefing interview we create the opportunity for the participant to reflect upon the user experience as a whole and also come with input on how to improve it.

We used a PC running GNU/Linux Ubuntu 17.10 with the latest build of ECDAR 2.0, including projects designed for the test, and a smartphone (iPhone X) to film the evaluation for later references.

The tasks, we asked the participants to perform, have been made such that the participants get to explore the *must have* features (from Table 1.2), but also some parts of the editor, in order to get a better understanding of ECDAR 2.0. The six *must have* features from the MoSCoW are covered in three tasks, where the participants get to:

- #1 Explore the model and initial simulation,
- #2 Query the system and explore the result,
- #3 Inspect and add values to a simulation.

The tasks given to the participants can be found in Appendix E. In order to balance against the carry over effect, the tasks have been shuffled such that the participants got the tasks in random order.

4.1.1 Task #1 - Explore the Model and Initial Simulation

In this task, we want to see if the participants can get an overall understanding of how the simulator works and how to interact with it. In this way, we are testing the usability of the following implemented *must have* features: *Feature #1 (Manual stepwise execution of simulation)*, *feature #2 (Overview of steps in trace)*, and *feature #6 (Go to initial state)*.

This task is an introductory task, where we want to see if the participant knows how to open an example project and can explain to us what components and which system it contains. The beginning of this task mainly involves the editor, since we want the participants to get an overall impression of ECDAR 2.0, and we would like them to discover the simulator by themselves (the other tasks start in the simulator).

We expect the participants to go to the simulator and perform a number of transitions, so they can understand the connection between what they saw in the editor and the system that is being simulated.

We would also like to know about the participants' understanding of the simulated system and its transitions, so unless they explain it by themselves, we ask questions about what effect a transition has on the system.

Lastly we would like to see if the participants can go back to the initial step in the simulation.

4.1.2 Task #2 - Query the System and Explore the Result

In this task we want the participants to create a query and get the simulator to show the resulting strategy of the query. This task covers the following *must have* features: *Feature #1*, *feature #4*, and *feature #21 (See traces from verifier in the simulator)*.

We start this task in the simulator, as we do not want the participants to explore the editor in this task. The first part of the task requires the participants to add a new query, which is done from the right-hand side pane, however, we have collapsed the query pane element, i.e. it does not show any existing queries, and the participants have to expand it to add new queries.

In this task we would also like to hear about the participants' understanding of the system. We may question the participants about what the current state of the system is and how a transition affects the system. Compared to the first task, we would also like to ask about the participants' understanding of the text string that represents a transition. Do they know what channel it synchronizes on, which process outputs, and which processes receive an input?

At the end of the task we ask the participants to further explore the trace, to understand the effect of the performed transitions, and reset the simulation to the initial state of the resulting query. It is possible to reload the complete simulation i.e. it removes the resulting strategy and fetches changes to the components from the editor. So we want to check if the participants understand the differences between reloading the simulation and going to the initial state.

4.1.3 Task #3 - Inspect and Add Values to a Simulation

In the last task the participants explore the following *must have* features: *Feature #1*, *feature #3* (*Inspection of state values*), and *feature #6* (*Go to initial state*).

The main goal is to see whether the participants understands how to perform transitions with a delay, and see how that delay affected the clocks in the system.

This task also starts in the simulator, but this time we use the zoom functionality to enlarge the processes (they cannot see all processes at once). We want to see if the participants notice that it is zoomed in, and what actions they perform to be able to see more processes at the same time. They might try to zoom out using a keyboard shortcut, look for zoom buttons in the *View* menu item, or perhaps close the right-hand side pane.

We first ask the participants to perform a transition with a specific delay and then inspect the processes to see if their clock values updated. Thus, we want to see if they know how the delay functionality works and whether they can locate the clocks.

We would also like the participants to perform a large delay, that is not allowed due to some guard conditions. In this case we want to see if the participants understand why the transition could not be performed and what the delay should be to make the transition legal.

Lastly we want the participant to add a new clock to a specific component. This part requires the participants to understand that they need to open the editor, locate the place where clocks can be added, and reload the simulator to see the updated system.

4.2 Results

From the evaluation, we identified 18 usability issues (listed in [Appendix F](#)). Some of the issues relate to the editor, since the tasks required the participants to also get slight knowledge about the editor. The issues from the editor have to some extent been inherited from the H-UPPAAL project, as we use it as the codebase of ECDAR 2.0.

We have rated each usability issue with a severity of either *cosmetic*, *serious*, or *critical*, based on Molich [12]. This severity is related to what the participants expected to happen and what actually happened.

Cosmetic issues have a small difference between expectations and what happened. The participant is able to quickly recover from these types of issues. An example could be the participant accidentally adds an extra query and is able to undo the action by deleting it again.

Serious issues have a significant difference between expectations and what happened. The participant might try different actions to recover, and we might interfere to ask questions about the task to help get them back on track. An example of such an issue could be that the participant cannot immediately find the simulator, and looks for it in different menu items. We would then ask questions about where they are currently located (in the editor or the simulator).

Critical issues have a critical difference between expectations and what happened. This type of issue might lead the participants to get stuck, such that they cannot continue

the task. We interfere when we can see that the participant has tried numerous actions and can still not proceed.

The issues presented in [Appendix F](#) have each been assigned a severity. For each participant we have assigned a severity to the issues they experienced. If the same issue is discovered by multiple participants, we assign the highest experienced severity to the issue e.g. if P1 experiences an issue as cosmetic but P2 experiences the same issue but as critical, we assign the critical severity to the issue.

In total we discovered 18 usability issues: 9 cosmetic, 6 serious, and 3 critical ([Table 4.1](#)).

Severity	# of issues
Cosmetic	9
Serious	6
Critical	3
Total	18

Table 4.1: The total number of usability issues identified

The rest of this section highlights all of the critical issues and the issues experienced by four or more participants.

4.2.1 Usability Issue #2 – Querypane Does not Expand When Adding a new Query

Severity: Critical Frequency: 8/9



Figure 4.1: The top of the querypane element

In task #2 we asked the participants to add a new query, but we did however collapse the querypane element, requiring the participants to expand it to see the queries for the system. [Figure 4.1](#) presents the top of the querypane element, where the chevron (leftmost button in the figure) can be pressed to collapse and expand the view.

Almost every participant (8 out of 9) pressed the plus button expecting it to add a new query, but to them it appeared as nothing happened. Some participants tried to press the button again, even more than once, and some looked for other places in ECDAR 2.0 where they could add queries e.g. looked for actions in menu items, right clicking on the query element, or going to the editor.

What actually happens when the participants clicked the plus button is that a new query was added to the querypane element but it stayed collapsed, so the participants

could not see that a new query was added. This issue was common among both the novices and experts.

A simple fix for this issue, is to simply expand the querypane when the plus button is clicked. This was also suggested by some of the participants, since they expected such behavior.

4.2.2 Usability Issue #3 – Hard to Know What is Being Simulated

Severity: Serious Frequency: 4/9

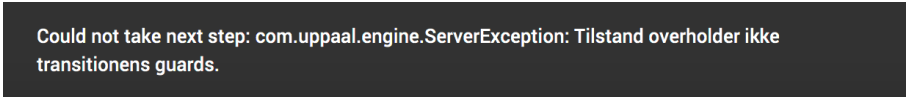
This usability issue is related to the participants' understanding of what is being simulated. There are two main types of simulations: System simulation and strategy simulation. The system simulation is a way to explore the declared system and its processes. The strategy simulation is used to inspect the resulting strategy from a query. In task #2 we wanted the participants to explore the strategy resulting from a query, so it is important that they know the difference between which type of simulation is currently being performed.

This issue was prevalent among the novice participants. The experts had a better understanding of which transitions should be available and could identify whether it was the strategy they were simulating.

A solution to this issue could be a text label at the top of the simulator, saying what is currently being simulated e.g. the label could contain something along the lines of "Simulating the initial system" or "Simulating query strategy". This could be combined with an indicator on the specific query that is being simulated.

4.2.3 Usability Issue #5 – Toast Message About the Guard Disappears too Quickly

Severity: Cosmetic Frequency: 8/9



Could not take next step: com.uppaal.engine.ServerException: Tilstand overholder ikke transitionens guards.

Figure 4.2: The toast when a guard is violated

In task #3 we ask the participants to perform a delay of 20.0 time units on a transition, which intentionally leads to a violation of a guard. The violation is presented to the participants by a toast message as the one shown in [Figure 4.2](#). This toast is a short informative message that disappears after about 3 seconds.

Before performing the transition two of the participants already understood that the transition was illegal, since they understood the process and its guards. Nevertheless, eight of the participants were confused by the toast and had trouble reading it before it was dismissed.

Some participants expected to find the message in the message container (among the warnings and errors), and others tried to perform the same illegal transition to force the message back on screen. One participant also noted that using the word "exception" in a message makes him concerned about the state of ECDAR 2.0 and he might consider restarting ECDAR 2.0 (*usability issue #6*).

The issue is only categorized as *cosmetic* since each participant was able to recover and understand the error, but it was observed frequently which makes it one of the top prioritized issues to fix (like the other issues highlighted in this section).

One suggestion a participant proposed was to add the message to the error/warning message container, which we think is an adequate solution. But we also believe there is a greater issue behind the current issue. Instead of showing an error message when the users try to do something illegal, we would rather avoid the situations where the users try so. Currently the users have to enter a delay and click a transition to see if the delay was legal, but we believe that ECDAR 2.0 should inform the users of legal delays before trying to perform a transition. This could for example be done with text on each transition showing the range of the possible delay or highlight the transitions that are possible when the users enter a delay. These are just initial suggestions for solutions, but we advise having a sketching session to generate new concepts.

4.2.4 Usability Issue #9 – Could not Find Values and Clocks

Severity: Critical Frequency: 7/9

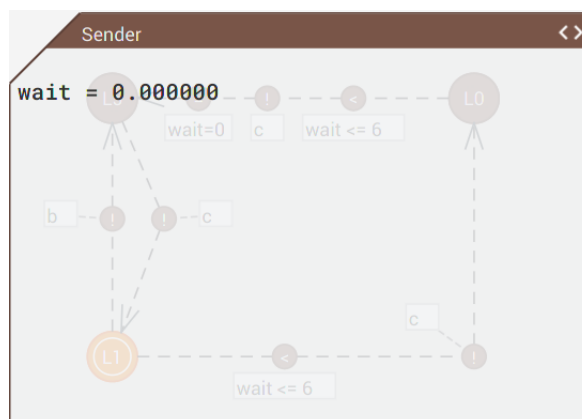


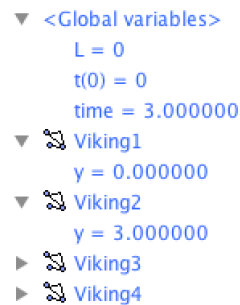
Figure 4.3: The values of the Sender process

In task #3 the participants are requested to show the clocks and values of the processes. Most of the participants (7 out of 9) had a hard time discovering where the values are located. To see values for a process, the participants have to press the <> button seen in the top-right corner of Figure 4.3. The participants who had used UPPAAL expected a pane with the values and clocks of all components. Six of the participants needed help with finding the correct location. The last one discovered it by himself through a

process-of-elimination. He guessed that `<>` was the place to click, because he could not find any other appropriate places. This is of course not how you want a user to process the interface.

Section 3.6 describes the design choices behind the placement of the state values and the consistency between the editor and simulator. It seems like the placement is counter-intuitive and that consistency does not matter when it is the first time they use ECDAR 2.0. Two participants mentioned that the icon used for the button did not make them think of values/clocks. The icon represents *code* according to the *Material Design* icon package we use, which might make sense in the editor but does not necessarily work in the simulator.

One of the expert participants suggested a solution like the one implemented in UPPAAL. The UPPAAL representation can be seen in Figure 4.4. He argues that being able to see all values and the process highlighting at the same time is important to his workflow. The UPPAAL representation is definitely a suitable solution for ECDAR 2.0, but we have concerns about how much space such a pane requires. We advise generating sketches for how to represent the values and clocks such that the user is able to see all values and highlighting at the same time.



```

▼ <Global variables>
  L = 0
  t(0) = 0
  time = 3.000000
▼ Viking1
  y = 0.000000
▼ Viking2
  y = 3.000000
▶ Viking3
▶ Viking4
  
```

Figure 4.4: The value and clock pane in Uppaal

4.2.5 Usability Issue #10 – Could not Find Where to Edit Clocks and Variables

Severity: Critical Frequency: 5/9

This issue was also discovered as a part of task #3 and it is related to *usability issue #9*. The participants had trouble finding the location where they could edit variables. As with *usability issue #9* the variables are placed behind a click on the `<>` button, this time from the editor. The frequency on this issue is lower than for *usability issue #9* which might be due to some of the participants having understood where the values are placed in the simulator. They were expected to edit values after seeing the values in the simulator.

Despite the design trying to be consistent between simulator and editor, we still observed the same problems as in *usability issue #9*. The five participants looked for the

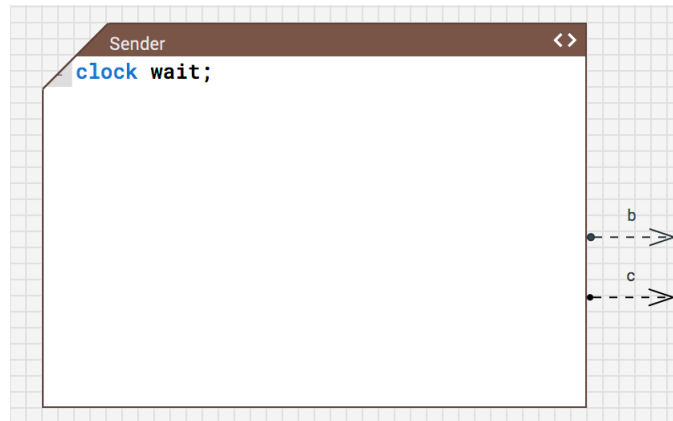


Figure 4.5: The editor for the Sender component

variables in many places e.g. pressing the *more* button on a component, right clicking a component, and looking for an action in the menu items. We do not have any good proposals for a solution, but we believe that the editor for component declarations should be visible at all times, when editing a component, and not "hidden behind" the components as the current implementation.

4.2.6 Debriefing Interview

As mentioned at the beginning of this chapter, after each session with tasks, we held a debriefing interview with the participants where they expressed what they liked about ECDAR 2.0, what they did not like, and their wishes for features or improvements. These interviews have led to positive feedback, wishes and also points of annoyance from the participants, which is summed up in this subsection.

Feedback

Besides all the usability issues the participants encountered they also gave us positive feedback on the design and user experience. Some of the features or design choices which they mentioned: The quick highlighting, the minimal design, keyboard shortcuts, and the universal location.

The quick highlighting of processes, locations, and edges, when hovering the mouse on either one of the available transitions or an element in the trace log, was mentioned by three of the participants as one of the features that gave them a much better overview of how the transitions affected the state of the system. One participant can be quoted as saying "I think this is a super cool way that the transitions are being shown. I am really satisfied with it".

The minimal design was mentioned by four participants as an important element in getting and regaining overview of a simulation. In general, the participants also mentioned that the overall design was modern and nice. About the overall design one

participant said "I think that the actual visuals are very nice and it is very easy to see what things are".

During the evaluation we also noticed that five participants used the keyboard shortcuts in ECDAR 2.0 to switch between editor and simulator, as well as to zoom and open an existing project. The keyboard shortcuts for zooming and opening projects are consistent with the operating system and many other applications, and the participants had an expectation that they would also work in ECDAR 2.0, without us telling them to use the keyboard shortcuts. The left navigation pane, where the participants can click to switch between editor and simulator, has labels for keyboard shortcuts (seen in [Figure 4.6](#)). We did not expect the participants to learn and use these keyboard shortcuts after such a short session, but we observed two participants who already used them.



Figure 4.6: The left navigation pane

When we asked the participants about what the universal location with the asterisk “*” edges meant they could fairly easy answer, that it represented all the possible channels handled by the given process. It should be noted that none of the participants knew about the concept of a universal location, but as they intuitively understood it, we see it as good design choice. This might due to the asterisk meaning *zero or more* in the field of regular expressions, and it is also used as a wildcard character. The universal location design was introduced in Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1].

Suggestions for Improvements and Features

Throughout the evaluation a number of suggestions for improvements and features arose. They can be seen in [Appendix G](#). Some of the suggestions are covered in relation to the usability issues and some we have already identified in the MoSCoW analysis ([Section 1.2](#)) but we prioritized them as *Should have* or *Nice to have*. One example is *feature #20 Keyboard shortcuts in simulator*, which we prioritized as *Should have*. While there are some shortcuts, ECDAR 2.0 does not make extensive use of them. One participant mentioned that he would like to step through the trace using arrow keys.

One of the expert participants suggested that the different views such as the available transitions, query and trace log could be "dragged out" from the main application; creating a window of the view instead. In this way, it would free up more space for the processes and then he could have the other views on a second monitor.

Using windows for the different kinds of views is something we already considered (example in [Section 2.1](#)) but decided to not pursue, partially because we wanted to avoid a simulator that is split into many windows. Having to manage these windows could be

quite annoying. The suggestion proposes the windows as an optional feature, that will make the user interface more flexible. It can be very useful at some workstations, with multiple monitors, to have the ability to manage application windows. We see this as a feature that caters to experts, which has not been our main focus (the focus has been the *must have* features), but it is definitely something that should be considered for future development.

4.3 Summary

In this chapter we have described how we have conducted the usability evaluation with nine people in total, which revealed 18 usability issues: 9 cosmetic, 6 serious, and 3 critical. The three critical issues are: The query pane did not expand when adding a new query, the participants could not locate state values in the simulator, and had the same problem locating the declarations in the editor.

Besides the critical issues we have also described the issues that were identified by four or more participants.

In the debriefing interview we got valuable feedback as well as knowledge about possible changes. Examples of these are to make the pane elements able to be "dragged out", and addition of more keyboard shortcuts.

5 Discussion and Conclusion

In this project we have designed and developed a visual simulator for ECDAR 2.0. Model checkers, like ECDAR 2.0, benefit from having a visual simulator that lets users explore and debug their models. The visual simulator in ECDAR 2.0 uses the engine of ECDAR 0.10 to advance in simulations and get resulting strategies from queries. This engine is however not well documented and is inconsistent in its error handling, which among other things means that the simulator has to call certain backend methods multiple times, just to get the expected output (described in [Section 3.1](#)). Despite these issues, the backend still provides the functionality needed.

The main challenge of developing a visual simulator for ECDAR 2.0, as introduced in [Chapter 1](#) and described by Mouritzsen and Jensen [5], is to take the output from the backend, e.g. states and traces, and present it in a user interface. We approach this challenge by designing and implementing a user-centered simulator. To ensure that we involve users in this approach, we base the requirements on what users expect from a simulator, create and evaluate sketches in iterations, and involve users in an evaluation of the implemented simulator. The next few paragraphs describe and discuss this process.

Requirements The first step of our approach was to collect a set of requirements for the simulator (see [Section 1.2](#)). We looked for existing research related to the development of a visual simulator, but could not find anything. This could indicate that not much research has gone into designing the user interface for model checkers and simulators, making this a novel area of research (see [Section 1.3.3](#)).

Since we could not find any related research, we researched numerous tools with visual simulators, so we could collect a set of features that are used in simulators. This collection of features was used to create a questionnaire (see the questionnaire in [Appendix B](#)), asking participants whether they think a feature is *important* or *not important*. We observed that some participants had difficulties answering the binary choice between *important* and *not important* e.g. if they did not completely understand the described feature they might set a mark between the two choices. Molich [12] mentions that there are in principle at least seven different answers to a yes/no question, like the one we asked. He recommends using a 5 or 7 step scale, since it covers the range of different answers and gives the participant a neutral option. The features proposed in the questionnaire are from different domains and visual simulators, and we observed that it could be difficult to answer some questions if the participant did not understand how the feature could be applied in their domain, leaving the question unanswered. Molich [12] also notes that with questionnaires you have no control over how questions are interpreted. We believe that the results from the questionnaire are still very useful for getting an insight into the important features for a visual simulator, however the results cannot be used for more detailed analysis and ranking of the features.

Based on the answers from the questionnaire, we prioritize 11 of the features using the MoSCoW method. The most important features are prioritized as *must have* (see [Table 1.2](#)), and we define those features as a Minimum Viable Product (MVP) for a visual simulator.

Design For the design process we chose to follow the design funnel presented by Buxton [2] (described in [Section 1.1](#)). This process has been great for generating concepts and explore alternative design suggestions. We used the MVP requirements as an input for the design funnel i.e. that the sketches we generate are based on the *must have* features. The sketches are generated through phases of *concept generation* and *controlled convergence* (described in [Chapter 2](#)). In the *concept generation* steps we generated many sketches which enabled us to explore a great variety of creative concepts. For each *controlled convergence* step, we evaluated the sketches, dismissing some and keeping others for the next phase. We evaluated the sketches using SWOT analyses, by ranking sketches according to design principles, and making different configurations of a complete visual simulator.

The result of following the design funnel is a sketch of a complete visual simulator, containing all of the *must have* features.

Implementation Having followed a thorough design process, with multiple evaluations of the design, makes the implementation unambiguous, since we have a clear design to implement. It was however needed to make new design choices during the implementation, since we realized some concepts were not satisfactory. An example, found in [Section 3.6](#), was how we wanted to represent state values. The design had it placed in the message container, but we realized that this proposal did not match the purpose of the message container. Instead of creating a new ad hoc solution, we could go back and pick one of the other concepts that has been evaluated during the design. This way we ensured that the implemented solution is based on our user-centered approach.

Evaluation An important part of our user-centered approach is to evaluate the implementation with participating users. We arranged a usability evaluation (see [Chapter 4](#)) with nine participants, so we could identify usability issues and get suggestions from the participants. [Table 5.1](#) presents the total amount of identified usability issues grouped by their severity. We can use these issues to evaluate what parts of the design did not work as intended. The critical issues we identified had a large difference between what the user expected and what actually happened. An example issue is related to how the participants inspect state values (see [Section 4.2.4](#)): Seven of the participants could not find the intended place, which required us to interfere and hint at where they should look. We also interviewed the participants, and received feedback on what they liked and did not like. They also had suggestions for how to improve the simulator, some of which could be considered in an updated MVP or MoSCoW analysis.

Severity	# of issues
Cosmetic	9
Serious	6
Critical	3
Total	18

Table 5.1: The total number of usability issues found

Conclusion To summarize the accomplishments of this project, we set out to implement a visual simulator for ECDAR 2.0. We approached the main challenge of making the visual representation of the simulation from a user-centered approach. The requirements for the simulator are represented as a list of features prioritized using the MoSCoW method, where we deem the *must have* features as a MVP for a visual simulator. We believe that researching and designing user interfaces for model checkers is a novel field of research, since we were not able to find existing research in this field. The design of the simulator is a result of a creative design process (inspired by Buxton [2]), where we generated and evaluated many concepts. The concepts that made it into the final design are a product of the user requirements and multiple evaluations. The implemented simulator satisfies the MVP, it contains all of the *must have* features, and is largely based on the final design from the design process. Lastly the simulator has undergone a usability analysis where we identified 18 usability issues, and also received feedback and suggestions for improvements from the participants.

6 Future Work

With the implementation of the simulator in ECDAR 2.0, we have taken a big step towards making ECDAR 2.0 a complete IMVE. Currently, ECDAR 2.0 consists of IMVE features to make it easier for the users to work in the domain of TIOA, system views to help the users get an overview of how the system is composed, a visual simulator, and mutation based testing (introduced by the other two authors of [1]). The vision for ECDAR 2.0 does not just stop here and there is plenty of room for additions and features, which the rest of this chapter will cover.

6.1 Improved Text Editor for Declarations

In the current version of ECDAR 2.0 the text editor for writing declarations only supports some syntax highlighting, and not features like auto completion, syntax checking, and renaming, which are common in many IDEs. Implementing such a text editor into ECDAR 2.0 would be another great step to make ECDAR 2.0 an even better IMVE, and would also improve the users' productivity.

6.2 Usability Issues

In [Chapter 4](#), a total of 18 usability issues were identified by the participants of the usability evaluation. We would advice that the critical and most frequent issues should be looked at before a public release. Since these are major issues that could ruin the user experience of ECDAR 2.0. Suggestions for solution to most of the features are described in the [Section 4.2](#). The critical and frequent issues should be prioritized, but the serious, and to some degree, the cosmetic issues are also worth fixing.

6.3 New Engine for Ecdar

As described in [Section 3.1](#) we have faced quite some problems with the current backend when trying to utilize the simulation functionality. Due to these problems we would suggest to either fix the issues with the ECDAR 0.10 backend, or create a new backend for ECDAR.

A minor problem with the ECDAR 0.10 backend (as well as for UPPAAL TIGA) is that it is not freely available for non-academics (it is also closed sourced) and that it seems like the development and bug fixing of the tool has been halted.

If a new backend were to be made, we would suggest that it is dedicated to the theory of TIOA, and is kept open sourced, under the same license as the frontend of ECDAR 2.0 (MIT license). In this way the people using the backend, as well as the frontend of

ECDAR 2.0, would be able to configure, bugfix, and in other ways contribute to the backend. Besides that, it would also be easier to distribute and start using ECDAR 2.0, as non-academics would not need to get a license for ECDAR 0.10 in order to have a fully working model checker and IMVE.

6.4 Future Work from the Previous Semester

In Bartholomæussen, Gundersen, Lauritsen, and Ovesen [1] we also describe a number of suggestions for further improvements of ECDAR 2.0. There are still features we would like to see implemented for *system views*, but they are not included in this section. The rest of this section is heavily based on [1]. These are some of the features that we still believe are relevant suggestions and improvements.

6.4.1 Multiple Engines

The frontend of ECDAR 2.0 uses the backend of ECDAR 0.10 to perform verification. Contrary to ECDAR 0.10, the model checker PYECDAR¹ features robustness analysis and explicit computation of conjunction, composition, and quotient. PYECDAR could be added as an additional *choice of engine* for ECDAR 2.0, such that when writing a query, users could choose what engine to use for that query. Furthermore, if other compatible engines exist, we could add those too, for example a new engine for ECDAR (see Section 6.3).

6.4.2 Version Control Systems

Another suggestion is to include support for Version Control Systems (VCSs) such as Git and SVN. An extension like this could change the way that the users are collaborating on modelling projects today. The process of working with models, described by the interviews in [1], reminded us a bit about a plan-driven process. Furthermore, it seems to us that the process to get a model accepted is tedious and lengthy. As TIOA already affords stepwise design e.g. through refinement, a VCS extension could move the process of making these models towards a more agile modelling process.

An integration with a diff tool is also proposed in [1]. The diff tool could be able to display changes from one commit to another, and also run the same queries on the different versions of the same model.

6.4.3 Integrated Modelling and Verification Environment

In this section we list some of the features mentioned in [1], that could further assist in modelling and verification. For the full description of the features we advise reading section 10.6 of [1]. We believe the following features would improve the IMVE experience: Continuous syntax checking, model refactoring, additional background analysis, automatic generation of system declarations, and cloning of components.

¹<https://project.inria.fr/pyecdar/>

Refactoring is a common practice that many modern IDEs support through convenient features like extracting methods, renaming, etc [13]. It would be interesting to see how refactoring could be applied to the modelling domain. Some of the suggested features are also very simple, such as the cloning of components, which would make creation of similar components quick and easy.

Bibliography

- [1] Bartholomæussen, C. M., Gundersen, T. R., Lauritsen, R. M., and Ovesen, C., *A new integrated modelling and verification environment for compositional real-time systems: Ecdar 2.0*, Aalborg University, 2017.
- [2] Buxton, B., *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann, 2007, ISBN: 9780123740373.
- [3] Pugh, S., *Total Design: Integrated Methods for Successful Product Engineering*. Addison-Wesley Publishing Company, 1991, ISBN: 9780201416398.
- [4] Mouritzsen, N. K. and Jensen, R. H., *Introducing hierarchies to networks of timed automata - huppaal - a new integrated development environment for model checking*, Aalborg University, 2016.
- [5] —, “Improving the model checking activity using h-uppaal a new integrated development environment for model checking”, Master’s thesis, Aalborg University, 2017.
- [6] Ries, E., *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Publishing Group, 2011, ISBN: 9780307887917.
- [7] Benyon, D., *Designing interactive systems: a comprehensive guide to HCI and interaction design*, 2nd ed. Addison Wesley, 2010, pp. 152–156, ISBN: 9781447920113.
- [8] Aaen, I., *Essence – Pragmatic Software Innovation (Unpublished book draft)*. Aalborg: Department of Computer Science, Aalborg University, 2017.
- [9] Nielsen, J., *Usability Engineering*, 1st ed. Academic Press, 1993, ISBN: 0125184069.
- [10] Cohn, M., *Agile Estimating and Planning*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005, ISBN: 0131479415.
- [11] Object Management Group, *Omg unified modeling language*, <http://www.omg.org/spec/UML/2.5/PDF>, 2015.
- [12] Molich, R., *Usable Web Design*, 1st ed. Nyt Teknisk Forlag, 2007.
- [13] Fowler, M. and Beck, K., *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

A Tool Inspection

No.	Feature description	Found in
# 1	Manual stepwise execution of simulation	SPARX, CPN Tools, TAPAAL, UPPAAL, PRISM, visualState
# 2	Overview of steps in a trace (trace log)	SPARX, CPN Tools, TAPAAL, UPPAAL, PRISM
# 3	Inspection of state values (e.g. values of counters or markings in a CPN)	CPN Tools, TAPAAL, UPPAAL, PRISM, visualState
# 4	Go back and forth in trace	TAPAAL, UPPAAL, PRISM, visualState
# 5	Random execution of a simulation	SPARX, CPN Tools, TAPAAL, UPPAAL
# 6	Go to initial state (reset simulation)	SPARX, CPN Tools, TAPAAL, UPPAAL
# 7	Import / export trace	TAPAAL, UPPAAL, PRISM (only export)
# 8	Random choice of a single step	CPN Tools, TAPAAL, UPPAAL
# 9	Vary the simulation speed on execution of a random simulation	SPARX, TAPAAL, UPPAAL
# 10	Simulation breakpoints	SPARX, visualState
# 11	User-defined time delay (Choose specific (concrete) time delays when choosing transitions)	TAPAAL, UPPAAL

Table A.1: List of functional features

No.	Feature description	Found in
# 12	Perform n steps at once (Fast-forward (perform 50 steps at a time) or choose how many steps to perform (>1 steps))	CPN Tools, PRISM
# 13	Generate a gantt chart of a simulation	SPARX UPPAAL
# 14	Textual simulation feedback (e.g. why the simulation stopped or if simulation is time-consuming)	CPN Tools
# 15	Manually update values during simulation (e.g. changing markings, clocks etc)	CPN Tools
# 16	Stop criteria (halts the simulation when criteria is meet)	CPN Tools
# 17	Automatically shows the currently active model/component (if the simulator only shows a single model/component at a time)	SPARX
# 18	Move up or down in hierarchy (Step in/out)	SPARX
# 19	Token selection control (select random, oldest, youngest, manual)	TAPAAL
# 20	Keyboard shortcuts in simulator (e.g. “next component”, “delay and fire”, “delay one time unit”)	TAPAAL
# 21	See traces from the verifier in the simulator (e.g. after running a query)	UPPAAL
# 22	Activity diagram of the components in the simulated system	UPPAAL
# 23	Plot trace path into a graph	PRISM
# 24	Deterministic loop detection	PRISM
# 25	Fire events that are not enabled	visualState
# 26	Choose values to watch (e.g. add a variable to “watch” so it is easy to keep track of it)	visualState

Table A.2: List of functional features (cont.)

No.	Feature description	Found in
# 1	Select transitions in a listview	SPARX, CPN Tools, TAPAAL, UPPAAL, PRISM, visualState
# 2	View is updated with new information when user performs a step (no animation)	CPN Tools, TAPAAL, UPPAAL, visualState
# 3	Zoom	SPARX, TAPAAL, UPPAAL, visual-State
# 4	Resizable tabs	TAPAAL, UPPAAL, PRISM
# 5	Integrated simulator mode	SPARX, CPN Tools, TAPAAL
# 6	Separate simulator (e.g. simulator is in a separate tab or window)	UPPAAL, PRISM, visualState
# 7	Enabled transitions are highlighted in the model (may also blink)	CPN Tools, TAPAAL, visual-State
# 8	Draggable tabs / menus	SPARX, CPN Tools, visualState
# 9	“Greyed-out” elements and locations when they are inactive	SPARX
# 10	Select transition in a pop-up menu	SPARX
# 11	Customize look of the tool	SPARX
# 12	Select events/transitions directly on the model (no separate window)	TAPAAL
# 13	Selecting transition in the side (list of enabled transitions) makes the transition blink in the model	TAPAAL
# 14	Use colors to differentiate between transitions	TAPAAL
# 15	Shows all component instances used in a simulation (no need to change tabs or windows to see each component)	UPPAAL
# 16	Selecting (before performing) a transition show an updated view if the transition is performed	UPPAAL
# 17	Error messages are displayed on the model	CPN Tools
# 18	Related information is highlighted when e.g. selecting a place	CPN Tools
# 19	Customize color and look of current/previous location and transition	visualState
# 20	Configure simulator view (e.g. “render changes” or “render values”, change visibility of individual variables)	PRISM

Table A.3: List of user interface features

B Feature Selection Sheet

Feature description	Important	Not important
Manual stepwise execution of simulation		
Overview of steps in a trace (trace log)		
Inspection of state values (e.g. values of counters or markings in a CPN)		
Go back and forth in trace		
Random execution of a simulation		
Go to initial state (reset simulation)		
Import / export trace		
Random choice of a single step		
Vary the simulation speed on execution of a random simulation		
Simulation breakpoints		
User-defined time delay (Choose specific (concrete) time delays when choosing transitions)		
Perform n steps at once (Fast-forward (perform 50 steps at a time) or choose how many steps to perform (>1 steps))		
Generate a gantt chart of a simulation		
Textual simulation feedback (e.g. why the simulation stopped or if simulation is time-consuming)		
Manually update values during simulation (e.g. changing markings, clocks etc)		
Stop criteria (halts the simulation when criteria is meet)		
Automatically shows the currently active model/component (if the simulator only shows a single model/component at a time)		
Move up or down in hierarchy (Step in/out)		
Token selection control (select random, oldest, youngest, manual)		
Keyboard shortcuts in simulator (e.g. “next component”, “delay and fire”, “delay one time unit”)		
See traces from the verifier in the simulator (e.g. after running a query)		
Activity diagram of the components in the simulated system		
Plot trace path into a graph		
Deterministic loop detection		
Fire events that are not enabled		
Choose values to watch (e.g. add a variable to “watch” so it is easy to keep track of it)		

C Sketch Rankings

In [Table C.1](#) and [Table C.2](#) the sketches are separated into three partitions: High scoring , medium scoring (white), and low scoring (red) partition. The sketches in low scoring partition have a score ≤ 12 , the middle partition sketches have a score in the interval of $12 < \text{and} \leq 16.5$, and the sketches in the top scoring partition have a score ≥ 17 .

In the high scoring (green) partition are the following sketches: #1, #5, #9, #16, #21, #22, #26, #27, and #28.

In the medium scoring (white) partition are the following sketches: #4, #6, #7, #10, #12, #13, #14, #17, #18, #19, #20, #23, and #24.

In the low scoring (red) partition are the following sketches: #2, #3, #8, #11, #15, and #25.

		x 2	x 1.5	x 1	x 0.5	
Sketch no.	Feature	Affordance	Consistency	Feedback	Flexibility	Total
#1	#1	4	3	4	1	17
#2	#1	1	2	5	1	10.5
#3	#1	2	2	4	1	11.5
#4	#1	4	1	3	3	14
#5	#1, #2	4	4	3	1	17.5
#6	#2	4	2	3	2	15
#7	#2	2	4	3	2	14
#8	#1	2	2	3	1	10.5
#9	#1	4	4	3	1	17.5
#10	#1	3	2	3	1	12.5
#11	#1	1	3	3	2	10.5
#12	#1	3	2	3	1	12.5
#13	#2	4	2	2	1	13.5
#14	#2, #4	2	2	5	2	13
#15	#1	1	1	3	1	7
#16	#3	4	4	3	2	18
#17	#3	4	2	4	2	16
#18	#3	4	1	3	3	14
#19	#3	3	3	3	2	14.5
#20	#3	4	2	3	2	15

Table C.1: Sketch ranking table

		x 2	x 1.5	x 1	x 0.5	
Sketch no.	Feature	Affordance	Consistency	Feedback	Flexibility	Total
#21	#3	4	3	4	3	18
#22	#1, #2, #3, #4	4	4	3	3	18.5
#23	#1	4	3	3	2	16.5
#24	#1	4	2	3	1	14.5
#25	#1	3	1	4	1	12
#26	#1, #2, #3	4	4	3	4	19
#27	#1, #2, #3, #4	4	4	3	4	19
#28	#1, #2, #3, #4	4	4	3	4	19
#29	#1, #2, #3, #4	4	4	3	4	19

Table C.2: Sketch ranking table (cont.)

D Configuration Sketches

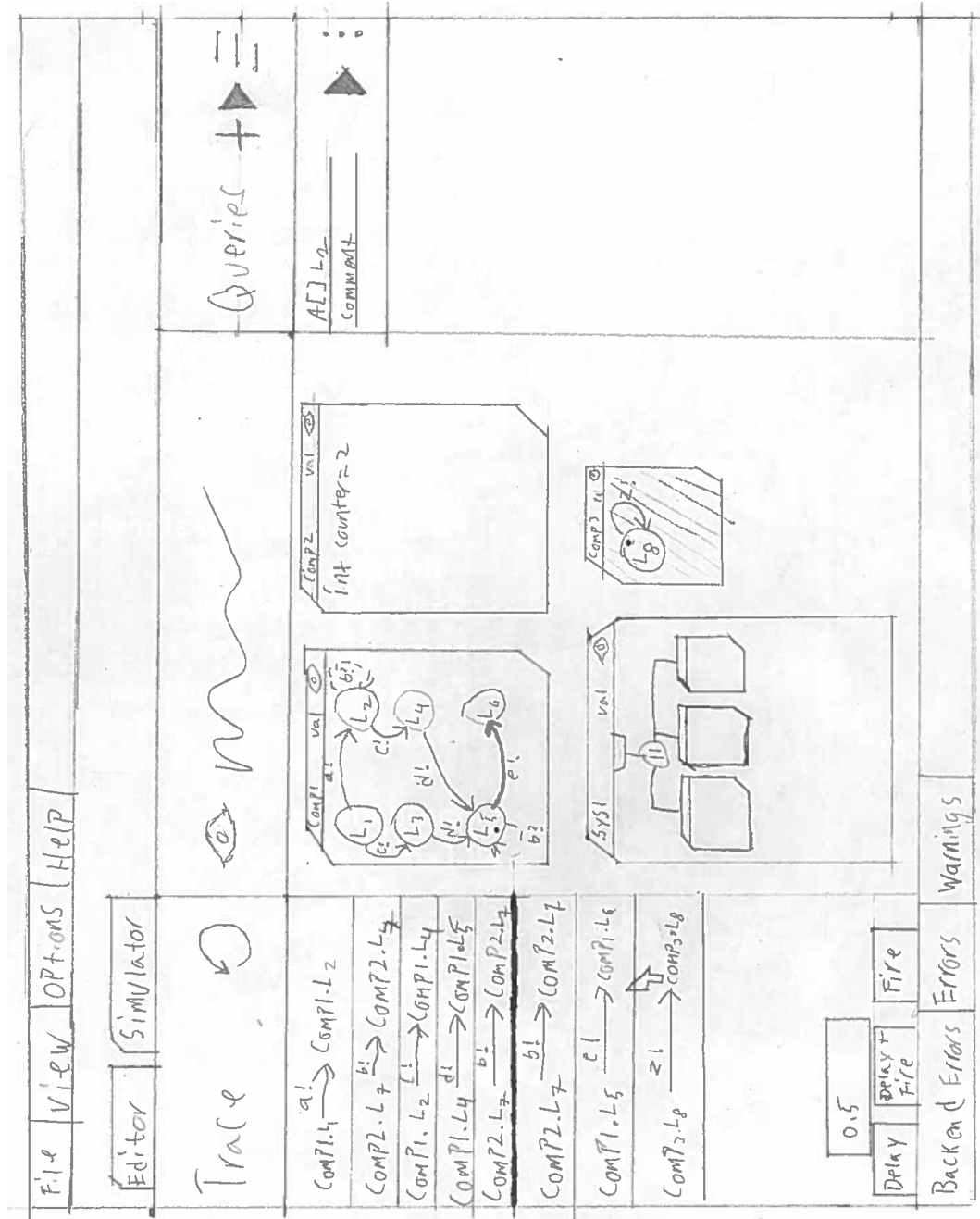


Figure D.1: Sketch of Configuration 1

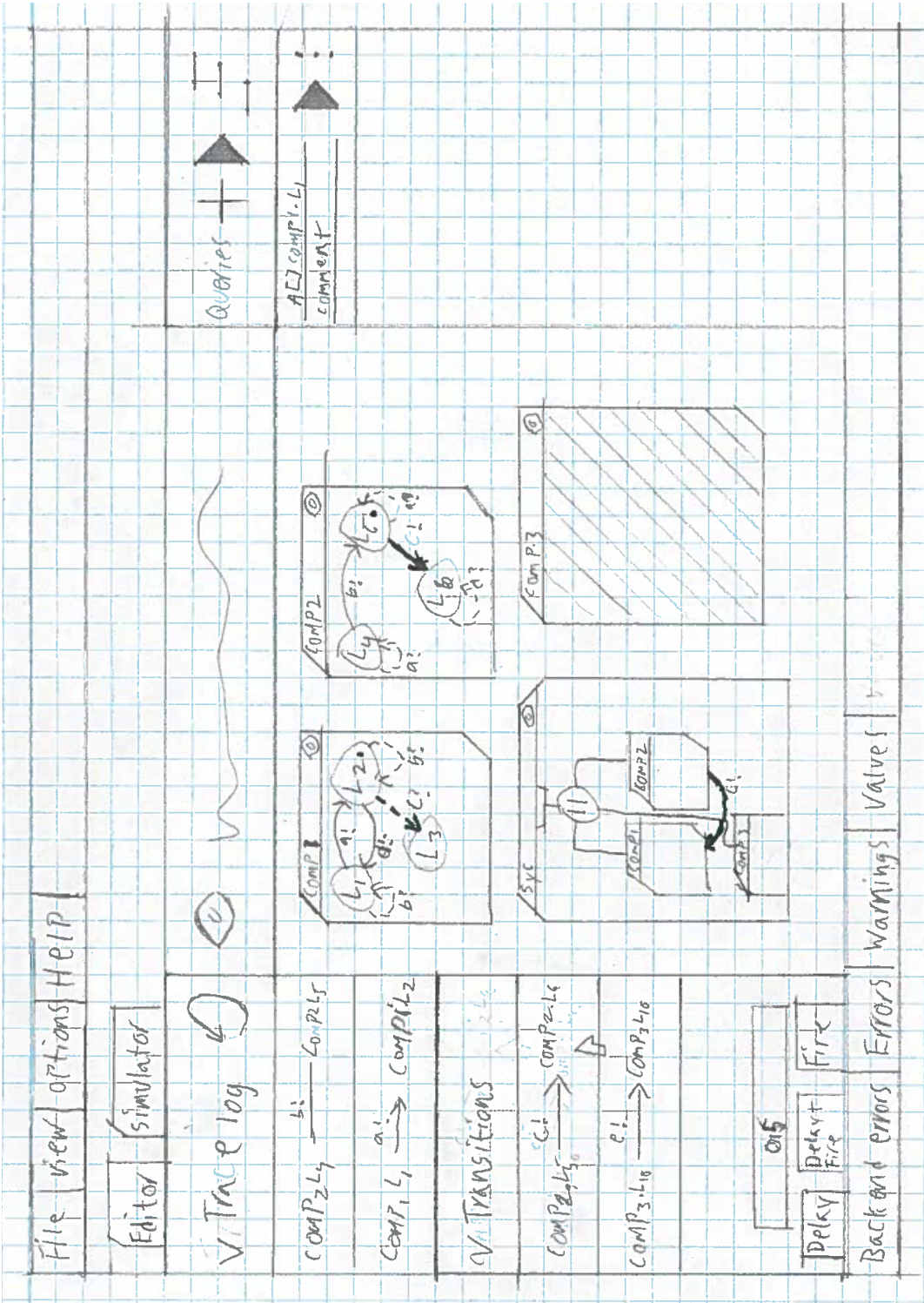


Figure D.2: Sketch of Configuration 2

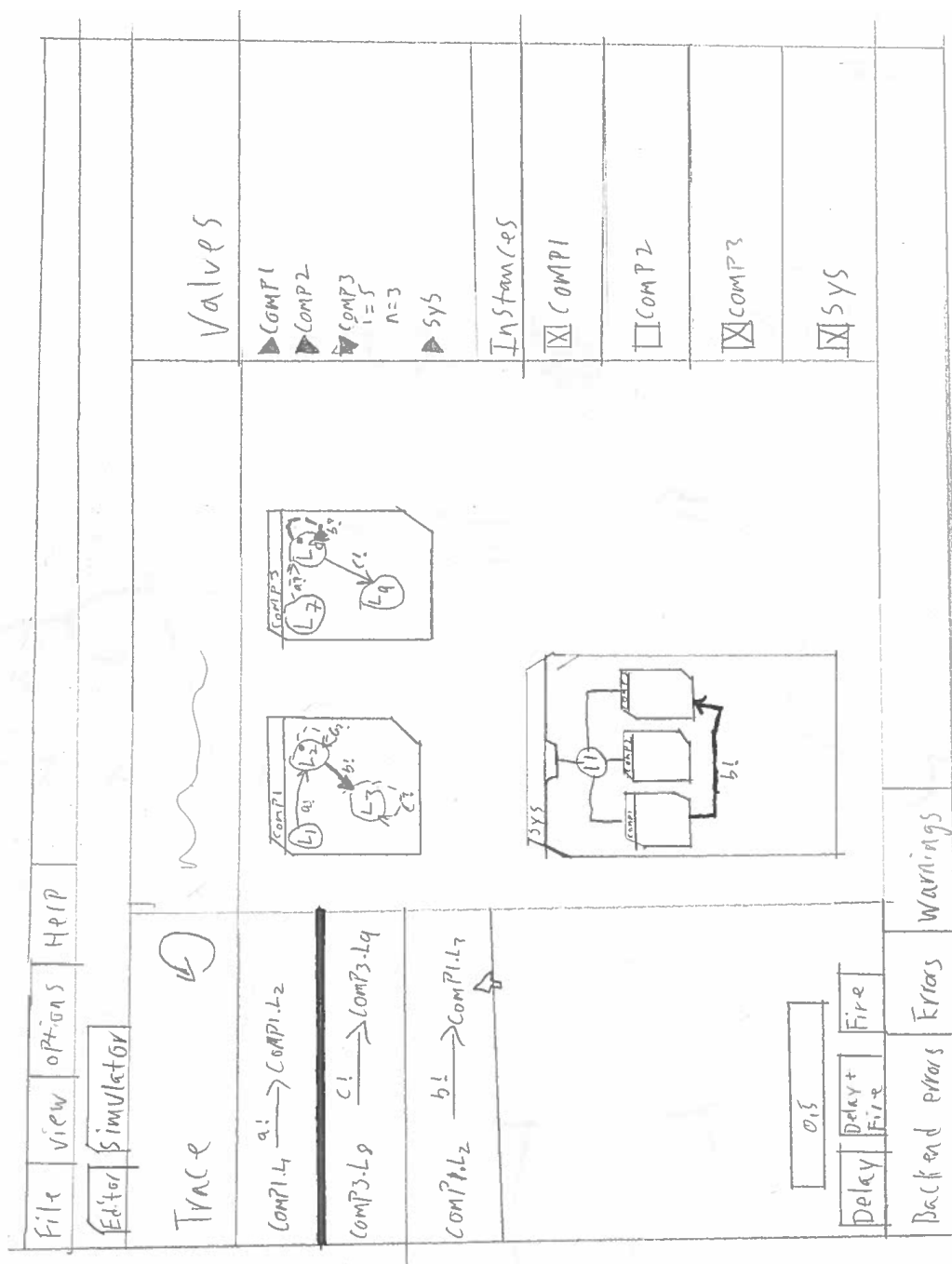


Figure D.3: Sketch of Configuration 3

E Usability Evaluation – Tasks

Task 1

Please open an existing Ecdar project, get a quick overview of the components and system declaration, and go to the simulator where you can explore the simulation of the system (no queries are needed). The project is the *AGTest* at `/samples/AGTest`.

While simulating the system, perform at least one transition that involves the *button2* channel. Feel also free to explore as many transition as you want.

When you are done with choosing transitions, revert the simulation to the state of the system where the processes are in the following locations L0, L2, L3 and L5.

Task 2

In this task we want you to create a new query (*specification: Receiver2*), and show the resulting strategy of the query in the simulator.

When the result is presented in the simulator, please consider what is the current state of the system and in which states the available transitions lead to (before actually performing a transition).

Please perform some of the available transitions (at least 5 transitions), and explore more of the simulator's functionality, such as the trace log.

At the end you should reset the simulation to the first state.

Task 3

In this task we want you to inspect clock and variable values during a simulation.

Firstly, please perform the “*c: Sender --> Receiver, Receiver2...*” transition with a delay of 2.0, and inspect how this transition affected the clock values of the processes.

Secondly, perform a transition with a delay of 20.0.

Lastly we want you to add a new clock *time* to component *Receiver2*, and go back into the simulator to inspect *time* during simulation.

F Usability Evaluation - Issues

Table F.1 and Table F.2 present the usability issues identified during the usability evaluation described in Chapter 4. Each issue has been assigned an identifying number, level of severity, and a frequency indicating how many participants experienced the issue.

No	Usability issue	Level of severity	Frequency
#1	Long loading time when opening a project	Cosmetic	P4
#2	Querypane does not expand when adding a new query	Critical	P1, P2, P4, P5, P6, P7, P8, P9
#3	Hard to know what is being simulated	Serious	P2, P5, P8, P9
#4	Expects the "run query" button to show the strategy in the simulator	Serious	P1, P4, P5
#5	Toast message about the guard disappears too quickly	Cosmetic	P2, P3, P4, P5, P6, P7, P8, P9
#6	Toast message containing the word "Exception" worried the participant	Serious	P4
#7	Values and clocks do not change while hovering the trace log	Cosmetic	P1
#8	Pressing a previous state in the trace log removes the rest of the trace until the selected point	Serious	P8, P9
#9	Could not find values and clocks	Critical	P1, P2, P3, P4, P5, P7, P8
#10	Could not find where to edit clocks and variables	Critical	P1, P2, P3, P6, P7
#11	Expect clicking on an edge to take a transition	Cosmetic	P2
#12	Hard to see differences between states when hovering on trace states	Cosmetic	P4
#13	Reloads the system when they need to select the first step in trace	Serious	P4, P5, P8
#14	Participant could not find "Show in simulator"	Serious	P8

Table F.1: The discovered usability issues

No	Usability issue	Level of severity	Frequency
#15	Cannot see the green acceptance color (and red) due to colorblindness	Cosmetic	P9
#16	Double clicks on the transitions (thinks the first click is selection)	Cosmetic	P2, P9
#17	Ordering of queries	Cosmetic	P4
#18	Hard to see which project is currently open	Cosmetic	P7

Table F.2: The discovered usability issues (cont.)

G Usability Evaluation – Suggestions

During the debriefing interviews of the usability evaluation (see [Chapter 4](#)), the participants suggested different improvements and features to ECDAR 2.0. Here we have collected an elaborate list of suggested improvements and features.

- Export a strategy
- Show allowed delays for a transition
- Better text for showing available transitions
- Add a tooltip if the transitions string is too long
- Add more keyboard shortcuts in the simulator (like TAPAAL)
- Add keyboard shortcuts for opening menu items
- Double-click on an edge to fire (first click should be selection)
- Undo/Redo buttons in the simulator
- Display values/clock such that processes could be shown in the same time
- Show a summary view when query pane is collapsed
- Support for multiple monitors (windowed panes)
- Export image of a simulation
- Make the application accessible for people with color blindness (red/green)
- Tooltip on nails (guard, sync., update, select)
- Universal location in trace log should be U# and not just Universal