Master Thesis

Robot Flow Control - Development of a Modular, Graphical User Interface

Author:
Matthias BÜTTNER

Supervisors:
Rico BELDER (host organisation)
Peter VON BALLMOOS (university)

August 31, 2015

Summary

At DLR, the Robotics and Mechatronics Center is developing a new software for robot flow control programming called RAFCON. It is built up from scratch and includes a lot of features coming directly from the people using the software after its completion. Therefore, a lot of requests regarding usability and functionality have to be taken into account. RAFCON uses hierarchical state machines, similar to Harel state machines, to implement the flow control, as this provides a good overview over the task and its parts and supports the step-by-step execution of encapsulated subtasks.

This work focuses on the development of a graphical, modular user interface simplifying the workflow and meeting the requirements. These are for instance the integration of RAFCON into the institutes corporate design, an intuitive user interface and fast state machine development. In order to create a consistent and matching design and color theme, the design of the GUI has been developed in collaboration with an external designer. The initial, bright color schema is replaced by a dark one, which reduces the stress on the human eye during work.

According to design theory, some important factors need to be considered when creating a user interface. The coloring and contrasts or "gestalt" principles are part of this. The colors of the controls (e.g. buttons, text outputs) need to be carefully chosen. Bright signal colors like red or orange guide the user focus on the important elements, whereas decent colors as blue or green are used to show uncritical information. Combining colors accurately, different contrasts can be created to further emphasize the importance of crucial events. Following "gestalt" principles, it is possible to arrange the elements in the GUI beneficial for the work flow and use them to generate a coherent look.

The created design orders and groups control-elements to improve the utilization of the available space and build a consistent user interface. New icons round out the appearance and help keeping the corporate design.

Furthermore, a new feature to rearrange single elements was implemented. It increases the possibility for the user to organize the control elements according to their needs. To gain more flexibility, this is used in the left and right side bars, providing additional information about the selected state and general information (e.g. state machine libraries, history). The logging view has been extended with a feature to choose which information (debug, info, warning and error) should be logged, which helps to improve finding important logging outputs. Additionally, a new toolbar is added next to the graphical editor, containing the execution control buttons (start, pause, stop).

Python is the chosen programming language for the entire project. Due to the predefined programming language, GTK is used for the window design and display. For the implementation of the main part of the GUI, the graphical editor, another library (Gaphas) has been used to gain more flexibility in customization. Gaphas is especially designed for the development and display of state machines. It is highly customizable regarding functionality and look.

All in all, the new design of the GUI could be realized successfully. The wanted effects of integrating RAFCON into the corporate design and improve the work flow could be achieved using the dark colored theme and the modular design. Furthermore, a lot of additional features have been implemented which work according to their needs. The new design and features turn RAFCON into a reliably working and intuitive software for the creation of robot flow controls.

Résumé

Au DLR, le centre robotique et mécatronique développe un nouveau logiciel pour la programmation du *robot flow control* appelé RAFCON. Il est construit à partir de zéro et comprend un grand nombre de fonctionnalités provenant directement des personnes qui vont utiliser le logiciel après son achèvement. Par conséquent, beaucoup de demandes concernant l'ergonomie et les fonctionnalités doivent être prises en compte. RAFCON utilise des machines d'état hiérarchiques, semblables à des machines d'état Harel, pour appliquer le contrôle de flux, car cela donne une bonne vue d'ensemble sur la tâche et ses parties et permet l'exécution pas à pas de sous-tâches encapsulés.

Cette thése se concentre sur le développement d'une interface utilisateur graphique modulaire simplifiant la productivité et répondant aux exigences. Ces derniéres sont par exemple l'intégration de RAFCON dans la charte graphique de l'institut, une interface utilisateur intuitive et un développement rapide de la machine d'état. Afin de créer un thème de design et de couleur uniforme et exact, la conception de la GUI a été développée en collaboration avec un designer extérieur. Le schéma de couleur initial, lumineux est remplacé par un schéma plus sombre, ce qui réduit la fatigue oculaire pendant le travail.

Selon la théorie de la conception, certains facteurs importants doivent être pris en considération lors de la création d'une interface utilisateur. La coloration et les contrastes ou les principes de "forme" font partie de ceux-ci. Les couleurs des contrôles (boutons, sorties de texte) doivent être choisies avec soin. Des Couleurs de signalisation chaudes comme le rouge ou l'orange guident la focalisation de l'utilisateur sur les éléments importants, tandis que les couleurs froides comme le bleu ou le vert sont utilisés pour afficher des informations moins importantes. En combinant les couleurs correctement, des contrastes différents peuvent être créés pour souligner davantage l'importance des événements cruciaux. Suivant les principes de "forme", il est possible de disposer les éléments dans l'interface graphique de maniére bénéfique pour le flux de travail et de les utiliser pour générer un regard cohérent.

Le conception crée trie et groupe les contrôleus pour améliorer l'utilisation de l'espace disponible et construire une interface utilisateur cohérente. De nouvelles icônes complètent l'apparence et aident à maintenir la charte graphique de l'entreprise.

De plus, une nouvelle fonctionnalité pour réorganiser les éléments a été appliquée. Elle augmente la possibilité pour l'utilisateur d'organiser les éléments de commande en fonction de ses besoins. Pour obtenir plus de flexibilité, elle est utilisée dans les barres latérales gauche et droite qui fournissent des informations supplémentaires à propos de l'état sélectionné et des informations générales (par exemple les bibliothèques de machine d'état, l'histoique). La zone de message a été étendue avec une fonction pour choisir les informations (debug, info, avertissement et erreur) à afficher, ce qui contribue à trouver les messages importants. De plus, une nouvelle barre d'outils est ajouté à côté de l'éditeur graphique, contenant les boutons de contrôle d'éxécution (démarrage, pause, arrêt).

Python est le langage de programmation choisi pour l'ensemble du projet. Comme le langage de programmation est prédéfini, GTK est utilisé pour la conception des fenêtres et l'affichage. Pour l'implémentation de la partie principale de l'interface graphique, l'éditeur graphique, une autre bibliothéque (Gaphas) a été utilisée pour obtenir une plus grande flexibilité dans la personnalisation. Gaphas est spécialement conçu pour le développement et l'affichage des machines d'état. Il est hautement personnalisable en ce qui concerne les fonctionnalités et l'apparence.

Dans l'ensemble, le nouveau design de l'interface graphique a pu être réalisé avec succès. Les effets recherchés de l'intégration RAFCON dans la charte graphique de l'entreprise et d'améliorer la productivité ont pu être obtenus en utilisant le thème de couleur sombre et la conception modulaire. De plus, un grand nombre de fonctionnalités supplémentaires ont été mises en place et fonctionnent selon les besoins pré-établis. Le nouveau design et fonctionnalités font de RAFCON un logiciel fiable et intuitif pour la création du *robot flow control*.

Contents

1 Introduction **1**

2 Theory **3**
2.1 Harel State Machines . . . 3
2.2 Design . . . 4
2.2.1 Four Gestalt Principles . . . 4
2.2.2 Colors . . . 5
2.2.3 Contrasts . . . 5
2.2.4 UI Controls . . . 6
2.2.5 Five planes of GUI design . . . 7
2.3 OpenGL and Gaphas . . . 8

3 Current Status **9**

4 GUI-Implementation **10**
4.1 GTK-Widgets . . . 10
4.2 Graphical Editor . . . 12
4.3 Features . . . 15
4.3.1 GTK-Widgets . . . 15
4.3.2 Gaphas . . . 16

5 Validation **19**
5.1 GUI . . . 19
5.1.1 Looks . . . 19
5.1.2 Functionality . . . 20
5.2 Problems . . . 22
5.2.1 General . . . 22
5.2.2 Known bugs . . . 23

6 Conclusion **24**

7 References **25**

List of Figures

1 Comparison graphical editor start/final 2
2 Simple state machine 3
3 Default state 3
4 Parallel state 4
5 Four Gestalt Principles 5
6 Selection of contrast types 6
7 Five planes of design 7
8 Tool at start time of designer GUI implementation in March 2015 9
9 Comparison between final versions of implementation (a) and designer (b) 11
10 Comparison between final state machine design 13
11 State view example 14
12 Connection view example 15
14 Debug view with different settings to show only needed logging output 16
13 Left bar widget 16
15 Different selection modes for data flow display 17
16 Differences state machine control bar widget 20

Acronyms

GTK GIMP Toolkit

GUI Graphical User Interface

HTML Hypertext Markup Language

OpenGL Open Graphics Library

CSS Cascading Style Sheets

DLR Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)

RAFCON RMC Advanced Flow CONtrol

1 Introduction

Robots become more and more important in, for example, medical [1] and home [2] applications. Because of the direct human interaction in both cases they need to be very precise and reliable to not harm any person. Therefore robots are required to have very accurate hardware, like high-precision actuators, but also a solid and reliable software implementation to control the robot properly.

On a low software-level, only simple commands, like "activate motor x" or "move wheel", are available, which need to be combined in upper-layer software to provide high-level *tasks* like "pick up item" or "move to position" [3]. This is done by combining low-level commands in a hierarchic manner to create higher level actions and tasks. For this purpose, the Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center) (DLR) started developing a new software for an easy robot flow control development called RMC Advanced Flow CONtrol (RAFCON).

This software is based on state machines, following a given order of commands to execute a predefined task. Createing such a state machine, without such software, the user types the whole code in a console window, which is a lot of work due to typing and keeping in mind the connections between the elements. Using this method only very simple tasks can be designed, as building more complex ones is very error-prone and hard to understand for other users not familiar with this particular state machine.

RAFCON helps the user by providing a Graphical User Interface (GUI) which visualizes and simplifies the construction of bigger and more complex state machines. This removes the need to remember the complexity of the skill and focus on the actual implementation. Most of the users are also more used to applications where it is possible to see the influence of an action on the work.

Objectives: The topic of this thesis is the development and implementation of the GUI in collaboration with an external designer. Firstly, the work included the design process with the designer, consisting of deciding the color scheme, the positioning and design of the items in the GUI as well as their functionality. This procedure was also used to check in advance if the design is viable with GIMP Toolkit (GTK), which is used in the entire project. Secondly, the implementation of the previous elaborated design into the software is the main part of this work. The implementation includes the decision about which framework to use for the display of the state machine in the context of RAFCON. Additionally, not only the graphical change needs to be implemented, but also the features needed to operate the GUI properly. At last, the implementation needs to be validated to make sure it works correctly and errors to be fixed.

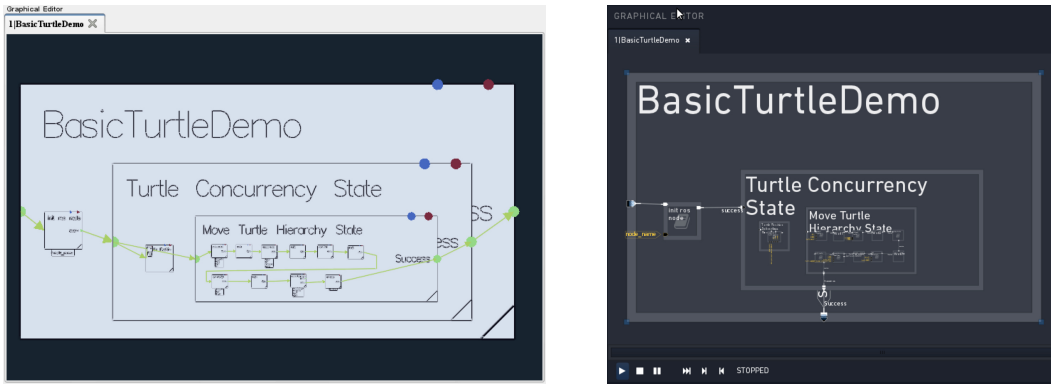
Methods and means: For the design process, several different design methods and patterns have been used and combined to use synergies between them. These are the four "Gestalt Principles" [4], which structure the design into proximity, similarity, continuity and closure. This helps to build a GUI which is steady and clear, in order to be intuitive for the user. Furthermore, coloring and contrasts, which build a unit can be used to create a lot of highlights and focus the users attention on important controls or events. The last method used for the design is the "five planes of GUI design", which provide a basic template on how to build a GUI from the strategy (the general scope) to the surface (what the user can see).

The actual implementation is completely done in the *Python* programming language, which provides the possibility of fast iterations, as the project does not need to be recompiled after every change. For the graphical representation, there are two tools in use: GTK for the display of the window itself and Gaphas [5] for the display of the state machine. GTK handles the interaction with the user on the screen and provides the possibility to build and display user interfaces using controls like buttons, sliders or lists. Even though it has a lot of features

not every part of the GUI can be displayed using GTK, which is why for the main element of the software a different framework called Gaphas is used. It is build on top of the GTK drawing context (canvas) and is especially designed to display state machines. Therefore some features can be used straight out of the box without any further work, which again increases the implementation speed of the design.

Structure: The work is structured as following: Chapter 2 explains the theoretical background in design theory, the difference between the Open Graphics Library (OpenGL) and Gaphas for the state machine display and provides an overview about Harel state machines, as well as the modifications of this approach. Chapter 3 shows the initial status of the implementation before the thesis, which leads to chapter 4 where the implementation of the new design is explained. After that the validation of the GUI implementation, the used design patterns and emerged problems are highlighted in chapter 5. At last, chapter 6 gives a short conclusion and perspective where the work on this project can be continued in future.

Results: In summary, the implementation of the new GUI for the RAFCON tool was accomplished successfully. Despite some problems during the realization which occurred because of unexpected compatibility issues with the used frameworks, GTK and Gaphas, the work could be continued after consultation with the team. Some differences in the final implementation compared to the designer template occurred because some features, which are already planned for the application, are not yet integrated and are therefore not considered in the latest realization. In figure 1, the graphical editor at the beginning of the thesis (1a) is put next to the final graphical editor (1b). The main difference is the coloring of the background, the port design and the coloring. Furthermore, the controls for the state machine is found directly below the graphical editor. For more detailed images to compare the initial, final and designer version refer to figures 8 and 9.



(a) Start (b) Final

Figure 1: Comparison graphical editor start/final

2 Theory

This part deals with a short introduction to Harel state machines and an overview of design patterns and their influence on user experience.

2.1 Harel State Machines

This sections is about the basic structure of state machines according to Harel. The main principles are explained with a few simple examples.

A state machine is a mainly event-driven system reacting to events and changing its status accordingly. Cars, planes or robots are examples for this kind of system. The following section is based on [6], if not specified otherwise.

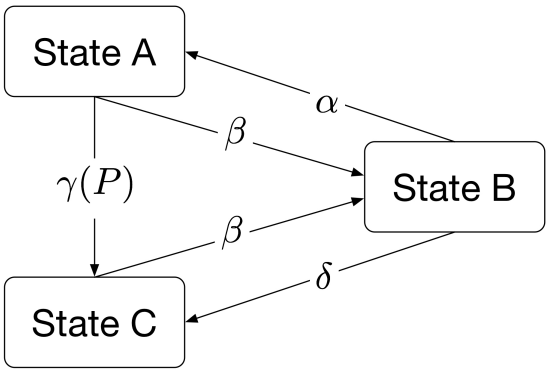


Figure 2: Simple state machine, based on [6]

A simplified example of a state machine is shown in figure 2, where there are three states and four different transitions between them. Assuming state A is active and event β occurs, the next state to be executed is B. If it was event $\gamma(P)$ with condition P the new active state would be C.

Depending on which state is active and which event occurs, a complex behavior can be mapped in a state machine. It is also possible for different states not to react on specific events at all, for instance if event δ occurs while state A is active, nothing happens as no transition for this event is set for A.

As state machines are pictured visually with transitions and states, their illustration simplifies the process of understanding even complex systems faster and with less effort. Looking at an image of a state machine directly shows the connections between different items and how the system is set up. On the other side, checking the plane code it might be difficult to understand the whole system with just a short glance.

A state machine consists of different state types being the *execution state* and the *hierarchy state*. An execution state is only responsible for the execution of the inserted code, whereas a hierarchy states' only responsibility is to coordinate several child states and to group them to a logical unit, for an easier overview of the state machine.

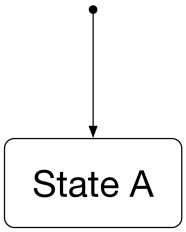


Figure 3: Default state, based on [6]

In figure 2, it is not obvious where the state machine starts as there is no indicator. To identify the start state, a new element pointing at the default state is inserted (see figure 3).

This is also needed when grouping several different states into one hierarchy state. To point out the start state the default state indicator makes it easy to immediately see, where the hierarchy state starts.

The default state indicator also helps creating parallel states, which are hierarchy states, executing multiple different state chains at once. If a state supports parallel execution there have to be multiple start states inside a container as shown in figure 4. The default start combination when entering state Y is B and F.

While executing a parallel state, there are two different possibilities to exit this state. One way of doing this, called "barrier", is to wait for all sub-states to finish and then exit the containing state, the other way, called "preemptive", is to interrupt all running states as soon as the first sub-state is finished [7].

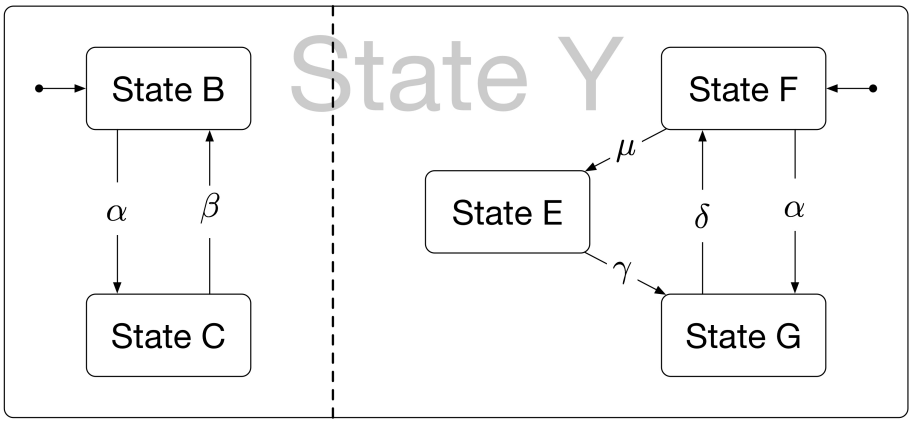


Figure 4: Parallel state, based on [6]

The state machines used in our project have some differences to Harel state machines. The most important one is that state machines created with RAFCON are not event-driven. Instead of events leading to the next state, a state has different outcomes which lead to the next state. Therefore, the state decides, depending on which result it obtains, which state to execute next. This leads to the next variance being incomes and outcomes. These two types of ports are used to logically connect the states with each other. An income is the entry to the states execution and outcomes are the possible exits of a state, where exactly one has to be chosen.

At last, the introduction of data flows improves the RAFCON state machines. Data flows are used to model the connections between states regarding their exchange of data, being calculation results or counters for instance. Every state can have zero or more inputs and outputs, which are used for the creation of data flows.

2.2 Design

Designing the GUI for a new application needs to be done very carefully. People tend to stop working with a program if it is unstructured and incomprehensible within the first couple of uses. Therefore, a solid and well implemented user interface is important for a pleasant ambiance for the user.

This section deals with some basic design principles to achieve this goal and to develop an application which is used gladly.

2.2.1 Four Gestalt Principles

The so called "Gestalt principles" are essential for building a GUI. They define a framework of how to arrange and group items in the interface. This part is based on [4].

There are four different, important gestalt principles, which are:

- Proximity: Items placed close to each other are recognized as group.
- Similarity: Using the same shape for different items associates them with each other.
- Continuity: The human eye looks for steady lines formed by small subitems for orientation.
- Closure: Groups of subitems are put into closed forms like rectangles.

In figure 5, the four principles can be seen in action. Each of them is important to create a whole picture which is internally consistent. An example is shown in the last image in this figure, which combines all four principles. It is easy to see that the design looks very tidy and easily understandable as the elements are grouped and aligned to each other.

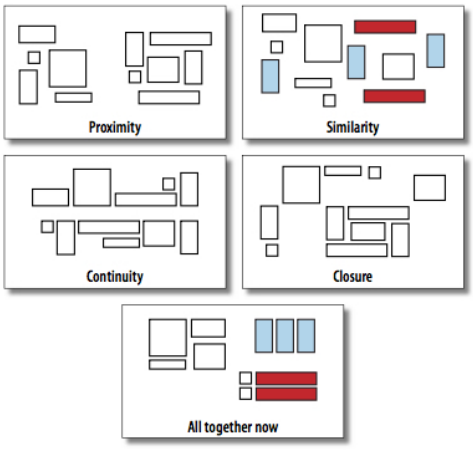


Figure 5: Four Gestalt Principles [4]

Using these principles is fundamental and helps structuring and planning the future design of the application. Especially in terms of long term support and integration of new functionalities, a good basic structure is essential for integrating new features and graphical elements (widgets) into an already existing GUI.

2.2.2 Colors

Making use of the different influences colors have on the perception of the user this is a very fundamental feature to guide and point to important upcoming events for instance. A lot of effects can be created using right color combinations in the correct spots. This section explains some basic examples of how to use this tool effectively. These paragraphs are taken from [8].

Using the same colors in different programs creates a corporate design and the user associates positive memories and contacts with the new product. But not only a positive association due to a corporate design is created by colors. Furthermore colors are used to grab the attention of the user, which can be used to draw attention to changes with the use of signal colors for example.

Bright and warm colors have a signal function on dark backgrounds and vice versa, signal colors, being red and orange for instance, can be used on bright and dark backgrounds to catch focus. Cold colors or colors complementary to the signal colors, like light green, are mainly used to put attention to subordinate information, which do not require immediate attention.

2.2.3 Contrasts

Contrasts are essential, too, especially when combined with expressive colors. This section explains a couple of different contrast types and which effect they attain.

In [9], which the next section and images are based on unless specified otherwise, the author lists several types of contrasts, which are for instance (i) Simultaneous contrast, (ii) Colored-uncolored contrast, (iii) Quality contrast or (iv) Bright-dark contrast.

In figure 6a all squares have the same green color in the center, describing the simultaneous contrast (i). Depending on the background color the green appears brighter/darker or more/less saturated. This is due to the different perception of the human eye according to the surrounding area. Using this technique it is easily possible to create a different feeling for the user by only changing the background color.

By applying a colored-uncolored contrast (ii) as shown in figure 6b the colors appear a lot more present and can be perfectly used to emphasize critical controls or popup windows. This is due to the fact that colored areas are very easily recognized on grayscale backgrounds [10].

The quality contrast (iii) in figure 6c shows the use of changing the saturation of one color. This method can be used to take items out of the focus without changing its actual color. Combining this method with the colored-uncolored contrast allows to dynamically move the users attention to a particular item. An example use case is moving the attention to currently active items by reducing the color saturation of surrounding items.

The last contrast listed here and shown in figure 6d is one of the most familiar contrasts, the bright-dark contrast (iv). It mainly helps pointing out details in the GUI. It is difficult and exhausting to differentiate between similar colors properly. Because of that it would not

(a) Simultaneous contrast

(b) Colored-uncolored contrast

(c) Quality contrast

(d) Bright-dark contrast

Figure 6: Selection of contrast types

be possible to see text or other UI elements, when the contrast is too low. In this context it is highly important to find a proper compromise between the readability, suffering at low contrast, and the tiredness of the eyes caused by a too sharp contrast.

Using multiple contrast types in combination allows to create a GUI that fits into the corporate design (if one exists) and at the same time to control the visual information flow to the important parts of the GUI.

2.2.4 UI Controls

In modern technology a large amount of different control possibilities exist. Due to the diversity of these controls the most important ones are explained and when they should be used or not.

Using [11] as source for the next section shows some different control methods, being (i) touch-screens, (ii) gesture sensors, (iii) smart pens and the classical (iv) keyboard and mouse control.

The standard keyboard/mouse control is used in most of the cases a desktop PC or Laptop is utilized. The interaction methods with a software are mouse clicks and movements, scrolling, hovering and typing with the keyboard. More up to date hardware offers more controls but it all can be subsumed to the previous mentioned controls.

According to which control should be supported by the GUI, different design aspects have to be taken into account. Using (multi-)touch screens for instance requires to have bigger icons and controls, as fingers are not as accurate as a mouse pointer. Therefore the complete GUI needs to be build with bigger buttons and handles, which need a lot more space than in common application designs. Some features intended for standard keyboard/mouse configurations like hovering, have to be disabled. This is due to the fact that with current technical capabilities a recognition of a finger hovering over a specific spot is not possible for example.

Touch screens also provide some features that simplify the control of a program. These are for example pinching to zoom or swiping to move a graphical viewer around. The latest generation of devices offer an additional degree of freedom using touch screens which is the force applied to the screen while pressing. This could be used to replace the lost hover ability to display for instance extra information [13].

As touch screens provide gesture recognition (swiping, pinching, pulling, etc.), a device which is capable of tracking and translating gestures made with your hands in free space to controls for programs has been invented. This quite new technology requires again different user interfaces to match the newly created control method.

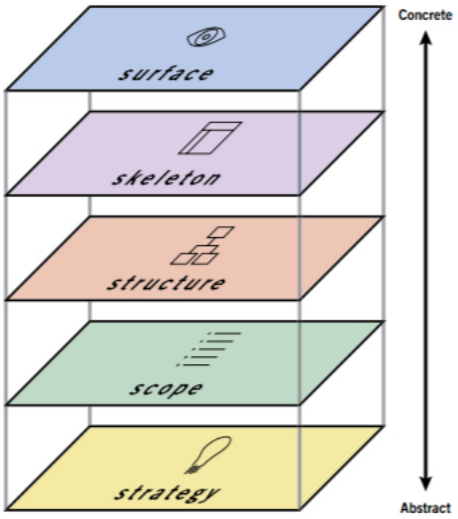


Figure 7: Five planes of design [12]

Another interesting control method aside the standard keyboard/mouse control is the use of smart pens, which can track and translate the movement of the pen into controls. There are a lot of degrees of freedom a smart pen provides apart from its position like pressure, angle or rotation. Implementing this control into a software does in fact not have big influence on the design itself but adds a lot of features regarding the control of the application.

2.2.5 Five planes of GUI design

Knowing the effects of coloring, contrast, controls and positioning and sizing it is possible to start creating the GUI. This section explains an approach of how to develop a working interface from scratch and how to structure every layer to get the best result.

Figure 7 shows an approach of how to structure the GUI from the very beginning, suggested by [12]. It consists of five planes, each one representing one step in the design process. The process starts at the bottom and continues to the top.

1. In the strategy plane the general scope is fixed. This includes the obvious purpose of the software as well as some implicit requirements. Like how is the software operated or what the user expects from the application.

2. In the scope plane the task is to define the scope in detail. This involves deciding the actual features necessary in the final application. Like saving content, adding items and so on.

3. In the structure plane, the overall navigation in the software is defined. This for example includes finding answers to the questions: "Where do I go when I click this button?" or "What happens when I move this slider?".

4. In the skeleton plane the result from the structure plane is used to design a final layout for the software. This includes the final position of widgets, buttons and other controls, arranging the groups to their final configuration, etc...

5. In the surface plane, the previous defined skeleton is filled with images, colors and text. This represents the result the user will be able to see and interact with.

Of course these steps should not be worked on independently, as there are many cross connections and each step is linked to its predecessor and successor. In some cases, a skeleton is

not applicable in the final surface implementation, because of long distances between subsequent controls, for instance. It is faster and easier to get to know these issues as early as possible, in order to fix them before putting too much effort into a design that is not working.

After finishing these five steps, including possible reiterations, the design process is finished and the GUI can be implemented in the software.

2.3 OpenGL and Gaphas

OpenGL is a powerful open source 3D graphics library which can be used for almost any purpose in displaying content on a screen. An object in OpenGL is constructed by providing a set of points with each x, y and z coordinate. Using this set the object is built using the simplest form with a surface, being the triangle, by each connecting three points with each other. Increasing the number of interpolated triangles improves the rendered image in 3D space but does not provide a huge improvement in 2D space.

Due to its nature OpenGL provides a huge degree of freedom, which can be used to create a fast rendered graphical display according to the specific needs of the software. But this great freedom also impairs fast development as for every new graphical element a lot of background work needs to be done. This is for instance the functionality, the interaction with other elements and so on. Additionally OpenGL does not provide a native support for displaying system fonts, which makes it necessary to include libraries taking care of the text rendering by drawing single lines forming the letters. The previous two paragraphs are based on [14].

The next part dealing with Gaphas is founded on the documentation of [5]. Gaphas, in contrast to OpenGL, is "only" a 2D library based on the GTK canvas. This reduces the freedom in at least one dimension but on the other hand it is a framework which is especially designed for displaying state machines.

It basically uses only two different types of objects to display the content, which are *Elements* and *Lines*, both inheriting from the base class *Item*. This provides a simple base for designing new objects for the canvas.

Making use of the "pango" library it is also possible to render any system font in the canvas. This helps customizing the graphical editor as fonts can easily be added and used inside the viewer to create a consistent look throughout the whole application.

Moreover a lot of features are already integrated and can be used right away. States for instance can be resized by dragging handles which appear in the corners while hovering the mouse above the state. Lines are provided with a simple waypoint feature allowing to add and remove waypoints without any further implementation to mention only a few of the features. In addition to that the view displaying the content keeps track of the hierarchical structure of the state machine as well as the mouse interactions with it.

Furthermore Gaphas provides a powerful constraint feature wich allows the user to easily limit the movement of items or force handles to move on a predefined path. This can be used to keep child states inside their parents for example or allow state ports only to move inside its border.

3 Current Status

The status at the begin of my work can be seen in figure 8. It shows the complete graphical user interface divided into six different areas. Area 1, later referred to as 'left bar', shows a tab bar containing global information, being:

- Libraries: Contains a tree view showing all predefined library states which can easily be entered into the state machine
- State tree: Contains a tree view listing all states inside the state machine ordered by hierarchy level and parent state
- Global variables: Holds a list of all global accessible variables
- History: Placeholder, as implementation had not been done yet

Area 2, later called 'graphical editor', displays the graphical editor which is the main part of the software. It allows the user to build and edit the state machine by adding, moving, resizing and connecting the states. During the state machine execution it shows the currently active state(s). The user may interact with the state machine in terms of changing code, moving states, add states and so on.

In area 3, the states editor is placed. This widget is responsible for displaying additional information about the selected states, which are for example its name, ID and type. Additionally, it shows a text editor to insert the code to be executed in this state, as well as two sections named data and logical linkage. The data linkage section contains detailed information about the data input and output ports and their connection to other states, whereas the logical linkage section contains detailed information about the connections between the states themselves.

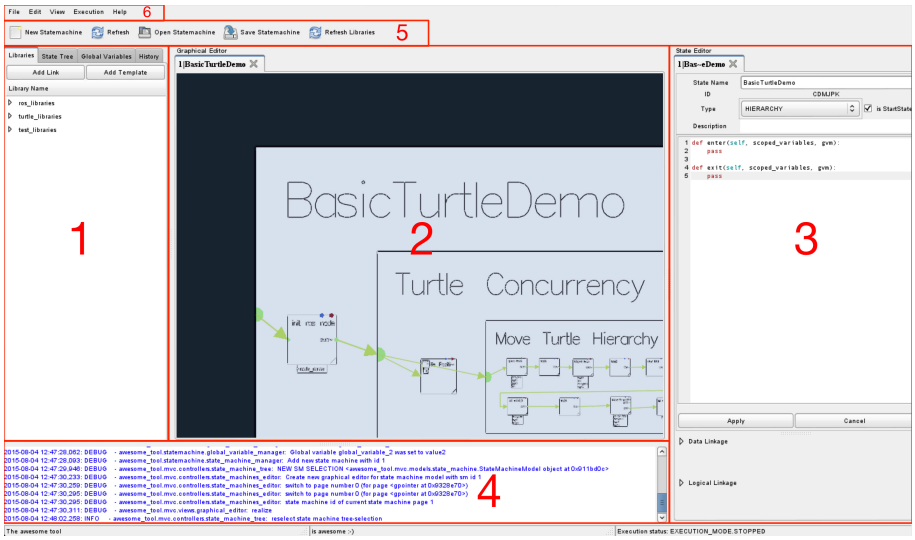


Figure 8: Tool at start time of designer GUI implementation in March 2015

Area 4, the 'logging view', is a simple text view holding all debug, information, warning and error outputs created by the program and the state machine (during execution). In this original state, it has no format and just displays all incoming messages without further filtering.

Above the main parts of the software, in the header part, there is a toolbar (area 5), called 'top toolbar', and menubar (area 6), called 'menubar'. The top toolbar basically holds some shortcuts to add a new state machine or load an existing one from the storage for instance. In the menubar all controls needed to operate the state machine and GUI are located.

4 GUI-Implementation

The GUI-Implementation focuses on the realization of the GUI provided by the designer.

The implementation of the GUI has been done in collaboration with an external designer. The initial drafts have been created by the designer and were discussed in the team developing the new software. This procedure ensured that usability and design fit and work together and to achieve a proper realization.

In the first design iterations the focus was exclusively on the widgets building the frame for the graphical editor. The graphical editor itself was designed in a second step, allowing easier and faster iterations. A detailed explanation of the implementation of the widgets, the graphical editor and the features is given in the next passages.

4.1 GTK-Widgets

The widgets were the first items to be redesigned. As the GTK framework offers limited possibilities for customizing the look, only few changes can be made. These are the colors and behavior while interacting with the elements and the size and positioning of the elements. A comparison between the final implementation and the final designer template can be seen in figure 9. Here the graphical editor is blacked out as it is not explained in this section.

The first step to implement the template, was the most obvious one, being the color change from light colors to dark ones. This is done by creating a GTK style file, holding the used colors and which widget is supposed to have which color. It is very important to specify in detail which item should be effected, as only default GTK elements are used. This is especially important when defining different colors for the same item (e.g. *GTKEventBox*, see listing 3). Listings 1 to 3 show excerpts from the created GTK style files.

Listing 1: GTK color definitions

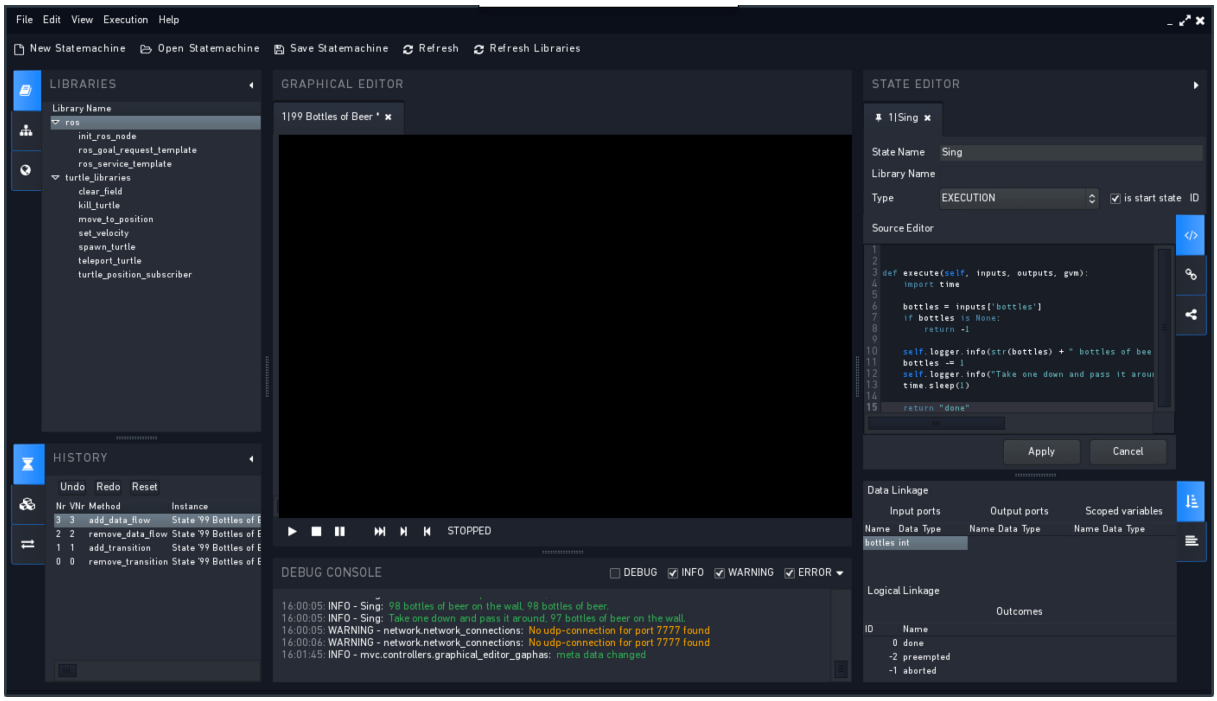
```
1 style "default"
2 {
3 ...
4 color["button_active"]                    = "#278bfe"
5 color["widget_background"]                = "#1d222c"
6 ...
```

In listing 1 the color definitions are set up in the style "default". Having these colors defined, a reference to a specific color can be made by using the @-identifier directly followed by the color name. Any color can be referenced using its HEX representation, of the three color parts R, G and B, "#ffeedd". This offers the big advantage of referencing all colors to one definition and simplifies changing the color, as not every representation has to be changed.

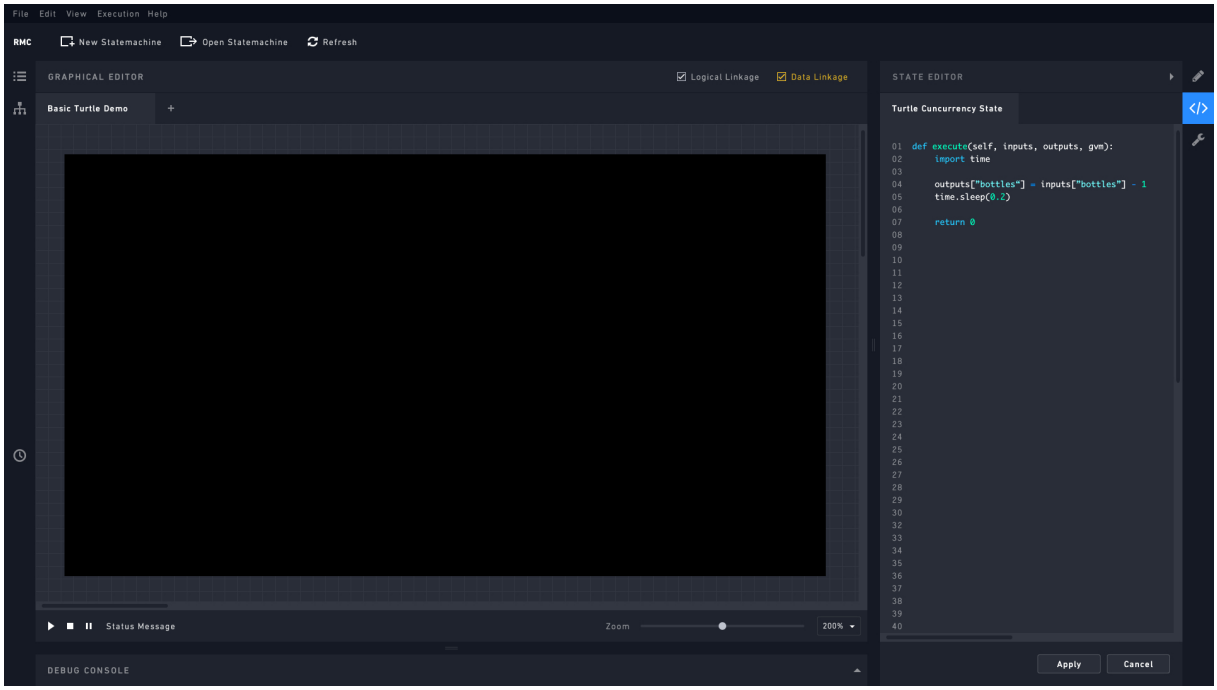
Listing 2: GTK button coloring

```
1 style "button" = "default"
2 {
3     bg[ACTIVE]          = @button_active
4     ...
5 }
6 widget_class "*Button*"              style "button"
7 widget_class "*ComboBox*"            style "button"
```

Looking at listing 2, a new style for buttons is created. The first line defines the styles name and its parent, being "button" and "default" respectively. The following lines inside the curled braces assign the colors to the elements of a button (like foreground (fg), background (bg), base and text), depending on the state (ACTIVE, NORMAL, PRELIGHT, SELECTED



(a) Final version of implementation



(b) Final version of designer

Figure 9: Comparison between final versions of implementation (a) and designer (b)

or INSENSITIVE). These different states are used when the state is the active element (e.g. clicked), simply displayed (no interaction like hovering or focus), prelighted (hovered), selected (e.g. check button is checked) or disabled. In case no color is set explicitly in the style, the default color, set in the "default"-style, is applied.

The last two lines assign the newly created style to the corresponding widgets inside the GUI. In this case *all* buttons and comboboxes receive the "button" style. This is due to the stars before and after "Button" and "ComboBox", which represent arbitrary placeholders.

Listing 3: GTK eventbox coloring example

```
1 style "top_label_event_box" = "default"
2 {
3       bg[NORMAL]      = @widget_background
4 }
5 widget_class "*VBox.GtkHBox.GtkVPaned.GtkVBox.GtkEventBox. ...
6                   ...GtkEventBox"           style "top_label_event_box"
```

In contrast to the previous example, listing 3 only applies to a very limited number of widgets. The style is applied to all widgets having the exact hierarchy as provided, being any VBox containing an HBox containing a VPaned and so on.

This procedure has to be repeated for every widget that needs to receive a custom coloring. For some widgets like notebooks, different styles need to be applied depending on their position and hierarchy inside the GUI. This is for example important when talking about the redesign of the left and right bar where the notebooks are supposed to have a blue background when selected but other notebooks don't.

Position and size of widgets are addressed next. The complexer widgets of the application are created with the Glade-3 tool for an easy "what-you-see-is-what-you-get" implementation. The output file of Glade-3 is a simple xml-file which can be loaded with builtin *Python* functions to create and build the GUI. This is a lot faster and less error-prone than implementing the whole structure in code by hand.

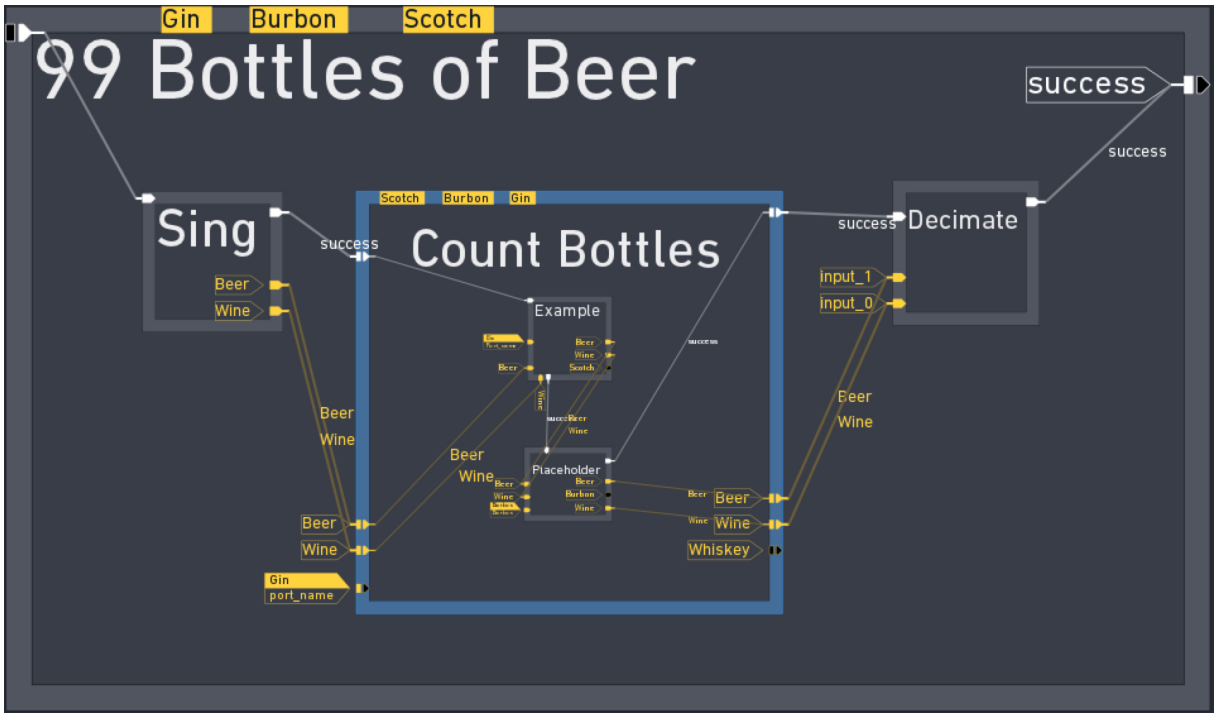
Furthermore widgets have the ability to send out signals on interaction. This can be a mouse click or release, as well as keyboard actions, while the widget has focus. These signals can be connected to functions where code can be inserted to execute when the signal occurs at the connected widget. Glade-3 offers the possibility to create these connections automatically. Moreover it provides the method name to be used inside the code in order to execute the needed action.

4.2 Graphical Editor

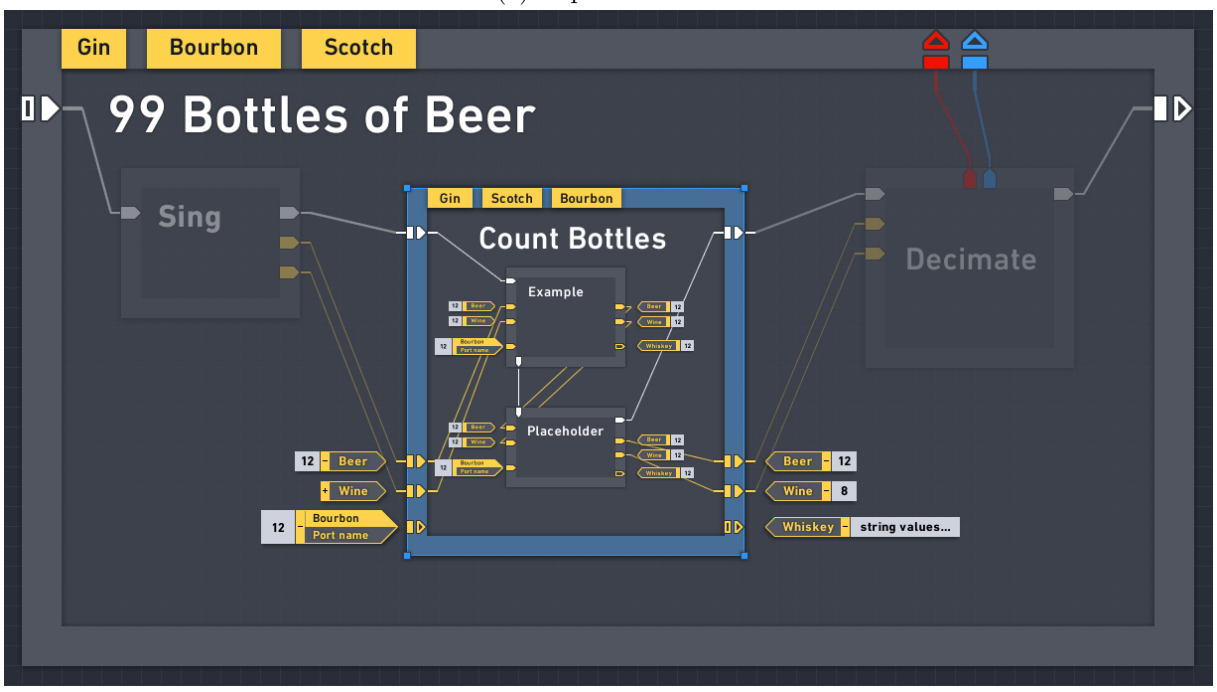
The realization of the graphical editor, which is the main part of the software, is a lot more complex than the implementation of the GTK widgets. First of all in the initial version the graphical editor was implemented using the OpenGL library presented in section 2.3. This provides a huge degree of freedom, but also limits the use of predefined elements as no framework for control features inside the editor is available.

A huge disadvantage of OpenGL is that there is no lightweight library built on top which could provide a solid foundation for visualizing and interacting with state machines. Therefore, an alternative to OpenGL needed to be found, offering some basic features for the graphical editor. After some research the choice was to use the GTK based library "Gaphas" [5]. A comparison between the final implementation and the template can be seen in figure 10.

The comparison in table 1 shows the advantages and disadvantages of OpenGL and Gaphas. For our purpose the possibility to use system fonts is more important than a possible future 3D release. This causes the graphical editor display to be switched from OpenGL to Gaphas.



(a) Implementation



(b) Designer

Figure 10: Comparison between final state machine design implementation (a) and designer (b)

Table 1: Advantages and disadvantages of OpenGL and Gaphas - Comparison

	OpenGL	**Gaphas**
Advantages	- All necessary features due to 3D environment - Powerful and performant implementation - Good documentation	- Already implemented framework for state machines - Easy customization of look - Use of any font installed in the system possible
Disadvantages	- No open source library for easy state machine display available - No support for system font display - A lot of work needed for implementation of new features	- Performance problems due to heavy background calculations (rarely) - Poor documentation in terms of feature customization (recognized after using it)

The change of the drawing framework allows to fully implement the given design from the very beginning and customize the controls and features to the new look. In the case of RAFCON there are two different types of items, being *State views* and *Connection views*.

State views are responsible for showing the single states inside a state machine. They contain the state name, its income and outcomes for the logical flow as well as its inputs, outputs and scoped variables for the data flow. Therefore inside a state view element, all these items need to be drawn.

This is done by maintaining a list of every port type inside the state view for easy iteration and drawing of the ports. The ports are stored in different classes inheriting from one central port-class providing all basic functions to simplify the differentiation between the ports. All port classes customize the look, direction and color of the port according to its purpose. The name itself is stored in a sub-item of the state view in order to make it resizable, movable and better customizable.

To be able to differentiate between the single ports of a state, a label pointing towards the port is drawn showing the ports name.

Additionally, to differentiate between selected, unselected and active states, there are three different colors to show the current status. A gray border is the default color for a non selected state and changes to blue when selected. In case of the state machine being executed the currently active states receive a green color.

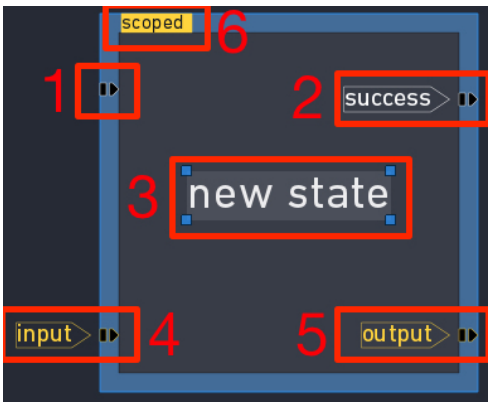


Figure 11: State view example

The new state view design, presented in figure 11, is split into a couple of elements for easier referenciation. Number 1 and 2 are logic ports with 1 being the income and 2 an outcome (there can be more than one outcome). As there is always exactly one income, there is no label showing its name. The outcome on the other hand shows its name to be able to differentiate between multiple outcomes. Number 3 displays the state name, which is its own object for better control. It shows a slightly brighter background when selected, as can be seen in the image. The ports marked with number 4, 5 and 6 are data ports, being an input, output and scoped variable respectively.

As ports can be moved along the state border, the label of outgoing ports (outcomes and outputs) are drawn inside the state, whereas incoming ports (inputs) draw their label on the outside of the state.

Connection views, on the other hand are responsible for displaying the logical connections and data flows in the state machine. They inherit from the line item, which provides the basic functionality to add waypoints, snap to ports and so on.

Figure 12: Connection view example (in this case it is a transition)

In the first iteration there are three different types of connections:

- Transitions in white color (for logical connections)
- Data flows in yellow color (for data connections between states)
- Scoped variable data flows in yellow color (for data connections between states and scoped variables)

Figure 12 shows a transition connecting the outcome "success" of "State A" with the income of "State B". When a port is connected its name moves from the label to the middle of the connection line and adjusts when new waypoints (blue squares) are added.

4.3 Features

Implementing the above mentioned graphical changes is only the first part of the integration of the new look. The second part is to provide the interface with the needed features in order behave as planned.

4.3.1 GTK-Widgets

There are a lot of new features connected to the GTK-Widgets. Due to the amount of small improvements and fixes, only two major features are explained in detail, being the redesign of the left and right bar, as well as of the debug view.

Left and right bar: One of the biggest changes in the GUI is the left and right bar. Previously, there has only been one horizontal tab bar in the left bar (see figure 8, area 1) and no tab bar at all inside the state editor in the right bar (see figure 8, area 3).

In the right bar there are four major parts being:

- State overview (area where the most important information about the state are displayed)
- Source editor (text area where the code to be executed while state is active is put)
- Data linkage (section where all data flows and ports are listed and maintained)
- Logical linkage (section where all transitions and outcomes are listed and maintained)

All areas except the state overview are put into a single tab each. This improves the readability and gives each section more space. Additionally, the description text field is moved from the state overview into an own tab as well.

On both sides the tab bars are placed vertically to be able to see all tabs at once without scrolling. This increases the visibility and simplifies the control, as everything is visible immediately. Furthermore, each side is split into two separate notebooks which can be resized according to the current needs.

(a) Debug view hiding the debug output for more visibity

(b) Debug view showing all outputs coming from the state machine

Figure 14: Debug view with different settings to show only needed logging output

The left bar shown in figure 13 keeps its previous tab distribution and receives an additional tab for the execution history. From top to bottom, the tabs on the left side are: *Libraries*, *State Tree*, *Global Variables*, *History* and *Execution History*. Using the grabber shown in number 1, it is possible to rearrange the splitting size of the notebooks to have always enough space to show the needed items.

Moreover, the tabs can be rearranged within their side. This means, it is possible to drag and drop the selected tab into any of the notebooks on the respective side, for instance one tab from the top left can be moved to the tab bar on the bottom left. This provides additional flexibility for the user, as the GUI can be set up according to the users need.

Additionally, every of the three widget groups surrounding the graphical editor received a white arrow shown next to number 2, which allows the user to collapse the respective area to gain more space for the graphical view.

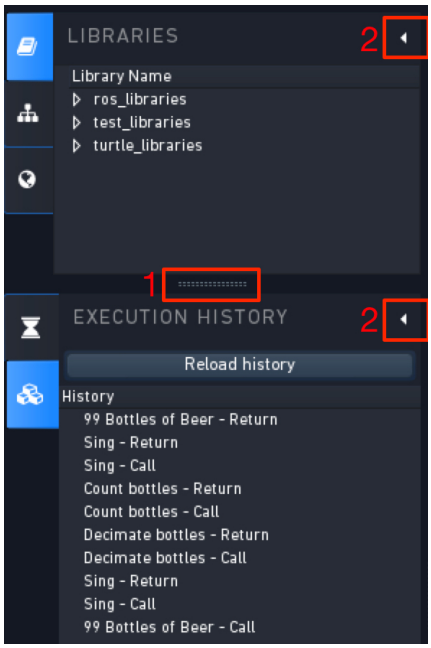


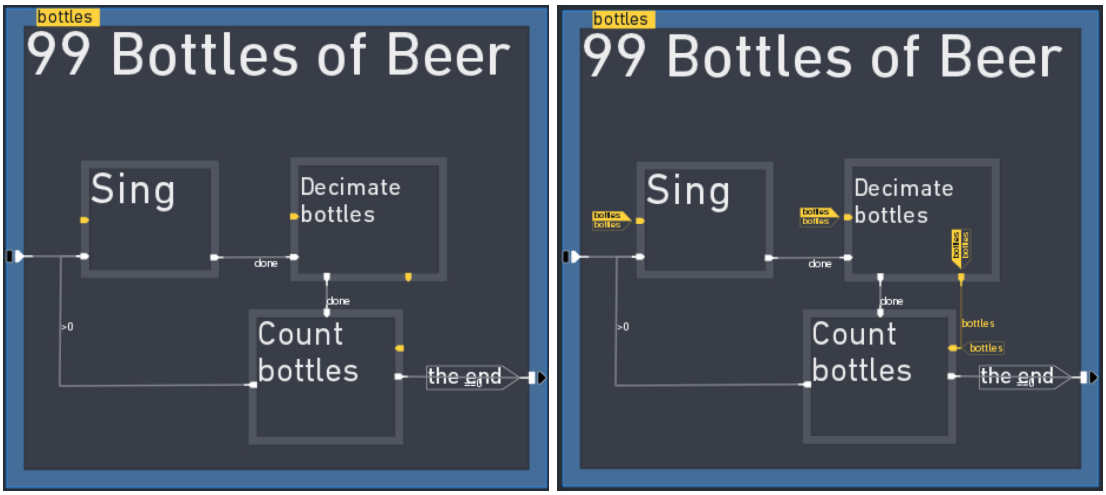
Figure 13: Left bar widget

Debug view: Another useful feature related to the GTK widgets is a small change in the debug view (see figure 8, area 4). This change allows the user to show and hide only the wanted logger outputs. By default, all information coming from the logger is displayed. However, it may be confusing to see all debug outputs when the state machine is running properly and the focus should be on the output of the state machine itself.

To make this possible the, improved implementation offers four checkboxes representing each of the four logger output levels: "debug", "info", "warning" and "error", which can be used to show/hide the corresponding messages. In the background, all messages are logged to an independent buffer, not to lose any logger outputs. Changing the selection, the complete buffer is filtered and only the selected messages are displayed in the debug view (see figure 14).

4.3.2 Gaphas

Switching from OpenGL to Gaphas is the biggest task, as a complete reimplementation of old features has to be done, on addition to the new features that were not available in the original version. The following passage explains some feature changes to the graphical editor.



(a) No data flows displayed

(b) All data flows displayed

(c) "Data flow mode" highlighting the selected state and its data flows

(d) Data port labels with value while state machine is executing

Figure 15: Different selection modes for data flow display shown on example of "99 Bottles of Beer" tutorial state machine

Data connections: Data connections are important in the GUI, as they handle the data transmission between the states. Despite being meaningful for the state machine, they are not required to be displayed all the time, as they tend to create a confusing net of lines and are not necessarily related to the logical flow. Because of that, it is possible to hide all data connections inside the graphical editor in order to improve the clarity. Three different actions for the data flows are possible:

Show/hide all data flows in the graphical editor: This feature is straight forward and is a simple shortcut and menu bar item setting a flag weather the data flows should be displayed or not. Only the data ports and the scoped variables in the states are not hidden to indicate their presence and weather the data ports are connected (filled). This can be seen in figure 15a.

Show only data flows connected to the selected state: In the case all data flows are shown, see figure 15b, every port draws its name and in case of connections to a scoped variable a split name label is drawn, which shows the name of the scoped variable in the filled background area and the name of the connected port in the non filled area. Also the data flows are displayed in the graphical editor using this setting.

Show/hide data flow values when the state machine is running: The third figure

15c shows the active data flow mode, where the selected state and its data connections are highlighted. This is realized by checking all available connection views, stored in the same hierarchy level as the selected state and its parent, if one end of them is connected to the state. In case this is true, a flag is set to *true* indicating the data connection should be drawn in the editor. Additionally all states except the currently selected one are set to be drawn with a transparency. This automatically focuses the attention of the user to the selected state and makes it easier to identify the data connections. When deselecting the state these flags are all reset to their default value, showing all states in full opacity and only display the data connections if the setting is active.

Finally in figure 15d the state machine is executed with the value label option turned on. This makes the graphical editor check each data port if it has a value stored and if so, draws this value inside a gray box next to the label. Given the case there is no value available yet at this port there is no box drawn at all until there is a value accessible. When a state machine is initially loaded from storage or newly created there are no values available at any port even if a default value is set. These values are evaluated when the state machine is executed. Therefore there are no values displayed in the beginning even when default values are specified, but the labels remain when the state machine was executed and stopped again. This is because the ports are not reseted during a state machine stop.

All these features in data editing support the user in order to be able to find the information needed as fast as possible.

Change transition/data flow: After creating a connection between two states, either a transition or a data flow, it should be possible to change the connection again without too many steps. In order to achieve this a feature helping the user changing connections is implemented. If the user wants to change a transition to restart state A after state A finished instead of starting state B, it is possible to simply drag the transitions end from B to A. This process needs to be supervised as a lot of actions lead to an invalid result and have to caught in order to restore the initial status.

This validity check, in the final version of this thesis, is performed inside the graphical editor when a connection is pulled to another port. This check should not be done at this point, because the graphical editor is not supposed to deal with this kind of operations. Therefore, after releasing the final version of this work, the connections themselves take care of their validity on every change, which makes it obsolete to check it in the graphical editor.

5 Validation

A verification of the changed GUI and if the new design corresponds to the theory is presented in this section. Moreover, a discussion of the problems and a possible solution is included.

5.1 GUI

In this section there are tests for its look and its functionality, as well as a verification if the design has been implemented according to the theory. The verification of the looks is done by manually comparing the template screenshots with the actual implementation.

5.1.1 Looks

The first validation, is the look of the GUI and if it is implemented successfully. In figure 9, the final version of the GUI and the designer template can be seen.

Altogether, the final implementation is close to the template. There are several differences which had to be made because of implementation problems, better usability or because not all features, designed in the template, have been implemented yet. Some differences and their source are:

- **State machine control bar:** The state machine control bar, which is located directly below the graphical editor, differs from the design template, see figure 16.

  First of all, there are more controls in the actual implementation than in the template. This is because there are more controls for the state machine control, than just *Play*, *Pause* and *Stop*.

  Secondly, the zoom bar is missing completely, as there were no fixed zooming guidelines at implementation time. Therefore, this feature is not implemented.

- **Icons:** The icons are different than in the template, as the designer used custom icons which were not yet available for the project during implementation. In an agreement with the designer, a temporal solution was found using icons from "Font Awesome" [15], which could easily be implemented.

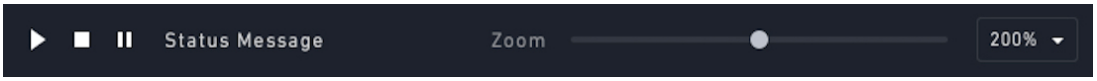
- **Ports:** To improve the visibility, the aborted (red) and preempted (blue) outcomes are not displayed at all by default. The user has to toggle a control in order to see all logic ports or only the white outcomes. In the design template, the *connected* aborted and preempted ports are drawn.

  Furthermore, the design template suggests a toggle button (+/-) for each port label to display the current value at data ports. This is not realized in this extend as the implementation of the control for each separate port is very time consuming and could not be finished. Therefore, the labels can be displayed either all at once or none (see section 4.3.2).

- **Right bar:** In the right bar, there is some difference to the template, too. These are the state overview shown at the top and the split into two separate notebooks.

  The state overview is a very essential part of the state editor as it contains its states ID, type and name. Hence, this widget is shown all the time for direct access to these important details. Directly below, the remaining state information is shown within two linked notebooks holding all tabs being the source editor, the data control, the logical linkage, an overview over all ports and the states description. The decision to split this side, too, was made due to better space management and adaptability.

(a) State machine control bar implementation



(b) State machine control bar designer

Figure 16: Differences state machine control bar widget

- **Graphical editor - "Data flow mode":** When in "Data flow mode" (see figure 15c), the root state is set transparent as well, in order to put the complete focus on the selected state and its data connections. In the design template, the root state also pulls a lot of focus, as it is highlighted too and appears in the same brightness as the selected state.

5.1.2 Functionality

The second validation deals with the newly implemented features and if they are functional. For some features, unit tests are used to verify if they are working, other workflows are tested by manually applying use cases, creating state machines from scratch or simply trying to crash the software by performing invalid operations.

The bugs found during these validations have been fixed as far as possible. Hence, a lot of errors could be resolved but also some harder to fix problems remain which are presented and discussed in section 5.2.2.

Creating a new state machine
To verify the functionality of the whole GUI, it is tested subsequently by hand, by building a new tutorial state machine from scratch. This aims at checking if the basic controls and features are working properly with the new GUI implementation. The verification is done by checking if problems and errors occur or not.

The single steps to build the state machine in this example are:

1. Create a new state machine
2. Add three new states to the root state and move the states to a new position
3. Rename the states
4. Add ports (outcomes, inputs, outputs) and scoped variables to states
5. Connect states logically and setup data flow
6. Add source code the states are supposed to execute
7. Run state machine

In the following, the above mentioned steps are explained in detail to show their purpose and which part of the implementation are tested.

The first step is to verify if the the "New Statemachine" button in the menu bar and shortcut tool bar still behave and work as before and if they change their background color when hovered and clicked. Furthermore, this step is to check if the root state, which is created with a new state machine, is displayed correctly.

After confirming the previous test did not produce any problems and errors, the next action is to add and move three new states. This is done by selecting the root state and using a shortcut to create the three states. During the root state selection it is tested if the state border changes its color to blue. Additionally it can be seen if the new states are inserted correctly into the root state. Moving the states triggers two effects: one is the selection of the state which causes the previous selected state to change its border color back to gray and the newly selected states border color to blue. The other one is, if during the motion a arrow cross appears, indicating the state movement. Finally in the states editor on the right side the tab showing the state information changes its content to the currently selected state.

Thirdly the states need to be renamed. This is done by clicking on a state to select it which opens the respective state editor on the right. In this editor the states name can be changed and is confirmed by pressing the enter key or clicking somewhere else. This applies the new name and changes the displayed name inside the graphical editor respectively.

Having this step verified the following action is adding the needed ports and scoped variables to the states. By selecting the state a new port or variable is supposed to be added, the right bar displays this states information. For adding a new data port for instance it is necessary to select the data linkage tab which shows the current data ports and connections. A new input is added by selecting the correct column and press the "Add" button. This causes a new input port to be added to the state, which is displayed in the bottom left corner of the state in the graphical viewer. Depending on the current data mode the label indicating the ports name is shown or not. In case the data mode is active the label holding the ports name is drawn and changes its content upon the port name change. Toggling this mode should make the labels appear and disappear. The same applies to outcomes, outputs and scoped variables.

With all states having the correct amount of ports and the ports being named properly the states can now be connected with each other. The first state to be executed inside the root state needs to be selected and set as start state inside the state overview or a new connection is dragged from the root states income to the desired start state. As an example the connection from the root state to its first child state is described in detail. First of all clicking and pulling from the income adds a new connection line starting at the port, which is now (partially) filled to indicate its connection to the mouse pointer. Moving the mouse over the destination states income port snaps the connection to it. Releasing the mouse button either when its hovered over the destination state or when the connection is snapped to the desired port finally creates the connection. If the start port of the connection is not an income the ports name is not displayed next to the port anymore but in the middle of the outgoing connection. This is also valid for other connections. Connecting a scoped variable behaves slightly different which can be seen in the disappearance of the connection and the appearance of the special name label mentioned in 4.3.2.

In the second last step the code to be executed in each state is filled in. As already known a state is selected to open its state editor on the right. In the source editor tab the source code is entered into the *execute()* method and needs to be applied with the "Apply" button.

At the end the state machine can be executed. While in data mode the values of the data ports can be shown or hidden using either the menu bar or a shortcut. Furthermore the currently active state is displayed with a green border to indicate the current execution progress. If logger outputs are inserted in the source code they can be seen in the debug views output. Additionally in the "Execution history" tab in the left bar new entries listing the previous execution steps appear with the currently available data values in a right click menu.

In case this test is working without any errors the minimum necessary controls to create a state machine with the GUI are functional. This method has been used after every major change and feature implementation to verify the basic functionality.

Trail and Error

For minor changes and features the trail and error tests have been used. In this case an already existing and saved state machine is loaded from storage and the GUI was tested if it crashes during usage. Given the case the application crashed, it was tried to identify and fix the error.

As an example the implementation of the copy/cut/paste feature was tested using this approach. First of all the new feature has been implemented, providing the start for the verification. After that, a state machine is loaded which already includes some states, that can be copied. One of these is selected, copied and pasted into the same parent or another container state.

In some cases is has been possible to copy the state without any problems. The problematic copy/paste actions occurred, when trying to paste the original state into a state, being smaller than the copied one. This problem was caused because of states not being resized. When trying to paste a big state into a small one, in comparison, the state and its children need to be shrinked accordingly. As there was no implementation taking care of that, the GUI crashed.

This resizing implementation itself needed to be tested independently with the "Trail and Error" method, to verify its functionality. After assuring the resizing mechanism is working properly in most of the cases, the verification of the copy/cut/paste feature can be continued. With this resizing algorithm, the pasting-action was working according to its purpose, as long as the target state was not smaller than factor four of the original state.

If the state was smaller than mentioned a not yet fixed problem of the resizing algorithm crashes the GUI. This problem is presented in detail in section 5.2.2.

5.2 Problems

This second section of validation deals with the problems that occurred during implementation and with bugs which could not be fixed because of time restrictions.

5.2.1 General

First of all, general problems during implementation are addressed and explained. This includes unimplementable design suggestions, feature requests which could not be realized and in this case limits that have been reached, while using the GTK2 and Gaphas framework.

In some cases, the design templates could not be realized because widgets and elements are not customizable enough in GTK version 2. Due to GTK2 only supporting the build-in GTK-RC files as color and spacing customization it has been difficult to implement every detail as in the design template. This could be solved by upgrading to GTK3, as this version supports Cascading Style Sheets (CSS), which provides a very high degree of freedom regarding customization of single widgets and items.

Furthermore the positioning in GTK2 is tricky as some hacks need to be applied to achieve the correct spacing in every widget. This is due to the concept GTK uses for adjusting the spacing between different widgets. It is only possible to set a border around a complete widget which scales down the entire widget, even if only one side is supposed to move further away from its neighbor widget. To achieve the wanted effect of solely adding some space on one side, a complicated structure of several different container widgets with specific settings have to be used. This increased the complexity of the entire GUI and did not work for every case as in some particular instances the complicated structure caused placement errors during runtime, which can hardly be debugged.

For the graphical editor where the Gaphas framework is used, few design features could not be realized as planned. For instance during the implementation of the name labels for data ports the interaction to show the current value by clicking on a small button next to the label

could not be integrated. As the structure of Gaphas, in this case, does not allow direct access to the element which drew the hovered label, it is not possible to implement an clickable button for this particular element. Therefore this feature had to be realized using a global menu item and a shortcut to toggle it for all data ports.

5.2.2 Known bugs

In this section, remaining bugs of the Gaphas implementation, in the graphical editor, are mentioned. These bugs occurred because of fast implementation cycles and a complex behavior that needed to be mapped.

One of the biggest known issues has already been mentioned in section 5.1.2, which is the GUI crash while resizing states. The bug happens during different actions:

1. **Resizing a state without its children:** When a state is being resized and the size becomes too small to fit the child states inside its body, the GUI crashes. This happens due to a child state having to stay inside its parent, in the RAFCON implementation. When the parent becomes too small to hold its children, the crash occurs as Gaphas cannot resolve the constraints anymore.

2. **Resizing a state and its children simultaneously:** During a simultaneous resize, for example the child states are resized with the same aspect ratio as their parent, the same problem as described in (1) occurs, when the child states reaches a size, where either their children or their border cannot be downsized anymore. To solve this problem a lower size limit of states could be implemented.

3. **Copying/cutting and pasting a state into a smaller one:** This process, described in 5.1.2, causes case (2) to take effect, too. Due to the pasting into a smaller target state, the copied state is resized and so are its children. This leads to possible crashes, when the copied state is downsized too much.

This bug could be solved by rewriting the downsizing algorithm and provide it with dynamic checks, which ensure there is always enough space for all child states to fit in their parent and prohibit a further decrease in size for the parent state. On the other hand these kinds of continuous checks can lead to a reduction of performance as they are possibly very resource consuming calculations.

The last issue mentioned here concerns the ports of a state. Due to the ports being part of the state, it is not possible to select a port explicitly. When clicking on a port always its parent state is selected, as the port is just another handle of the state. Therefore it is not possible to detect if a port is hovered or not for instance, as long as it is not clicked on. Making use of the tool-chain Gaphas provides, it is only possible to determine the port which is clicked on by receiving the clicked handle and check the selected state for this particular handle. This problem could be solved by inserting them as individual elements. Trying to change the implementation of the ports to be individual items, a lot of new issues occurred, which is why this realization has not been under further consideration in this thesis.

6 Conclusion

This thesis was set out to design and implement a graphical, modular user interface for a newly developed software with the help of an external designer. For the project the programming language "Python" is used. To integrate the GUI, GTK and Gaphas have been chosen to display the elements of the user interface and add its controls. With this task some questions needed to be solved, which are for example: "What is a good design to allow an easy and intuitive control?", "Which design theme should be used for relaxed working?", "Which features are needed inside a graphical editor, in order to be able to use the tool easily?"

Answers to these questions were found in collaboration with an external designer, who was in charge of planning the design of the new software, regarding the graphical elements and their potential use. Within several iterations the design was improved with ideas coming from our team as well as the designer to reach a final design that is functional, usable and good looking.

The result has been implemented as close as possible, which could be well achieved. Some small changes happened during programming, because of the used GTK framework or because a few features have been designated in the design template which have not been implemented yet.

It has to mentioned, the final version of the GUI, after this thesis work has been done, still contains some bugs which could not be resolved, due to the software being in constant change and a lot of new features have been added to the end of the thesis time. Therefore future work is to fix these bugs in order to provide a fully functional application.

In order to improve the flexibility in the design customization, the current used GTK version could be upgraded from two to three. This adds support for CSS and helps customizing the single widgets, buttons and other controls in the GUI. Moreover it receives frequent updates with new features and bug fixes, which helps making the application more stable and reliable.

Furthermore an integration of network control can be imagined. This includes a communication via network with a server handling a set of connected state machines and a display in a browser for multi platform and multi device usage of the software. A first step for this work has already been done in the scope of this thesis but is not mentioned in detail. This includes the implementation of a basic server, capable of sending and receiving messages via network from and to the tool as well as providing a web server to display the current status of the state machine in a browser.

In future this can be extended to fully support state machine generation, control and display. One possibility to achieve this is using GTK3's "Broadway" feature which allows to render the graphical output not to the screen itself but to a browser window, which allows direct control over all features without having to implement any additional functions. On the other hand a separate browser and touch screen optimized web interface can be developed for network control and surveillance of the state machines.

Both methods have their pros and cons regarding, implementation effort, usability, update time and so on. For instance with the "Broadway" feature it would be possible to simply continue developing the current design and automatically transfer it to the browser, at the cost of especially adapted controls for modern touch screens and other controls. This entire topic provides enough material for future works.

Besides that, another interesting topic for upcoming work is the development and implementation of an auto-routing algorithm for the graphical editor, which needs to be done all by hand in the current status of development. This feature would greatly improve the usability and would make it possible to display state machines which have been created automatically or without the help of the graphical editor.

7 References

[1] R. A. Beasley, "Medical robots: current systems and research directions," *Journal of Robotics*, vol. 2012, 2012.

[2] H. Lehmann, D. Syrdal, K. Dautenhahn, G. Gelderblom, S. Bedaf, and F. Amirabdollahian, "What should a robot do for you?-evaluating the needs of the elderly in the uk," in *The 6th International Conference on Advances in Computer–Human Interactions, Nice, France, February*, pp. 83–88, 2013.

[3] M. R. Pedersen, L. Nalpantidis, A. Bobick, and V. Krüger, "On the integration of hardware-abstracted robot skills for use in industrial scenarios," in *2nd International Workshop on Cognitive Robotics Systems: Replicating Human Actions and Activities*, pp. 1166–1171, 2013.

[4] J. Tidwell, *Designing interfaces*. O'Reilly Media, Inc., 2010.

[5] A. Molenaar, "Gaphas." `https://github.com/amolenaar/gaphas`, Mar. 2011.

[6] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[7] C. André, "Semantics of ssm (safe state machine)," *I3S Laboratory–UMR*, vol. 6070, 2003.

[8] T. Stapelkamp, *Interaction- und Interfacedesign: Web-, Game-, Produkt- und Servicedesign Usability und Interface als Corporate Identity*. Springer-Verlag, 2010.

[9] T. Stapelkamp, *Screen- und Interfacedesign*. Springer-Verlag, 2007.

[10] O. Jacob, *Konzeption und Gestaltung von Management Dashboards*. Hochschule für Angewandte Wissenschaften Neu-Ulm, Hochschulbibliothek, 2011.

[11] R. Dorau, *Emotionales Interaktionsdesign*. Springer-Verlag, 2011.

[12] J. J. Garrett, *Elements of user experience, the: user-centered design for the web and beyond*. Pearson Education, 2010.

[13] E. Betters, "What is force touch? apple's haptic feedback technology explained." `http://www.pocket-lint.com/news/133176-what-is-force-touch-apple-s-haptic-feedback-technology-explained`, Mar. 2015.

[14] R. S. Wright, B. Lipchak, and N. Haemel, *OpenGL superbible*. Addison-Wesley, 2007.

[15] D. Gandy, "Font awesome." `http://fortawesome.github.io/Font-Awesome/`, Mar. 2015.