FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Friedrich-Schiller-University Jena
Department of Informatics and Mathematics
Institute for Computer Science

Development of a Webtool for the Calculation and Analysation of Chemical Organizations

for the attainment of the academic degree of
Bachelor of Science (B.S.)

Presented by Fionn Daire Keogh
MatNr. 176219
Born March 18 1997 in Lörrach

1. Supervisor: Prof. Dr. Peter Dittrich
2. Supervisor: Prof. Dr. Christoph Kaleta,

Jena, October 16th 2021

Abstract

In order to tackle the problem of calculating chemical organizations, the web tool OrgWeb is presented in this thesis to tie in with the work of the offline tool OrgTools. The presentation of the tool includes not only the algorithm used, but also deals with the libraries used and the user interface.

The implemented algorithm used is based on the *Constructive Algorithm*. The *Constructive Algorithm* is one of currently two known fully computational algorithms. The heuristics are not discussed here.

OrgWeb is a good alternative to OrgTools. The lack of complicated installation is a big advantage over OrgTools. This could lead to a broader application of chemical organization theory.

Zusammenfassung

Um das Problem der Berechnung von chemischen Organisationen zu anzugehen wird in dieser Arbeit das Web-Tool OrgWeb vorgestellt. Damit wird an die Arbeit des offline Tools, OrgTools, angeknüpfen. Die Vorstellung des Tools umfasst nicht nur den genutzten Algorithmus, sondern behandelt auch die genutzten Bibliotheken und das User Interface.

Der implementierte Algorithmus basiert auf dem *Constructive Algorithmus*. Der *Constructive Algorithmus* ist einer von aktuell zwei bekannten vollstndig berechnenden Algorithmen. Die Heuristik wird hier nicht behandelt.

OrgWeb ist eine gute Alternative zu OrgTools. Das Fehlen einer komplizierten Installation ist ein grosser Vorteil gegenüber OrgTools. Dies könnte zu einer breiteren Anwendung der Chemischen Organisations Theorie fhren.

Contents

1. Introduction

1.1 Objective of this Thesis

Understanding the world we live in is a complex task. Not only since the information age have things been rapidly changing. To abstract the observed things mapping these to a systems is a common strategy. These systems can then be analysed and maybe a process or the function of a protein could be sensed.
The used systems – in classical systems theory – are static and do not account for novelty.[32] To account for novelty the Chemical Organization Theory was derived from Fontana and Buss' work[16] in the early 2000's.[12]

The aim of this thesis is to present an easy to use application for analysing chemical organizations. In the current time almost everyone knows what a web browser is or how to access a website. So providing a web application, with one code base to serve all three major desktop-systems and no installation of the application or dependencies, is one of the easiest methods to use a software-product.

This work will cover what the Chemical Organization Theory is, what Chemical Organizations are and how one would find organizations. Then I will give a brief overview of OrgTools (especially the Constructive Algorithm), what is needed for a web application and what tools were used. Chapter Two covers the implementation of the application, resulting in the presentation of the final product, followed by a discussion about limitations and what is left for the future.

1.2 Chemical Organization Theory

The Chemical Organization Theory tries to expands the concept of (biological-) organizations by Fontana and Buss to study constructive dynamical systems. Fontana and Buss define constructive dynamical systems as systems that display the production of novelty.[16] To understand what an organization is this Section will cover some basic concepts and definitions that are needed.

1.2.1 Concepts and Definitions

Algebraic Chemistry and Reaction Networks

Dittrich and Speroni di Fenizio[12] introduce the concept of an *algebraic chemistry*. 'Given a set $\mathcal{M}$ of elements [...] and a set of [...] rules $\mathcal{R}$ given by the relation

$\mathcal{R} : \mathcal{P}_M(\mathcal{M}) \times \mathcal{P}_M(\mathcal{M})$. We call the pair $\langle \mathcal{M}\mathcal{R} \rangle$ an *algebraic chemistry*'(Dittrich and Speroni di Fenizio, 2007, p. 1202 in [12]).

The term algebraic chemistry may be misleading, since this concept is not only usable for chemical computation, but has its applications in many fields of science. The components of an *algebraic chemistry* can vary in type. In one scientific application the elements of $\mathcal{M}$ are *molecules* and the rules of $\mathcal{R}$ are *reactions* between those *molecules*, as in a metabolism model of E. coli from Centler *et al.*, 2004[8], in another application the elements of $\mathcal{M}$ are *subject areas and disciplines* while $\mathcal{R}$ consists of *citations* and their creation of new scientific articles, Hegele, 2021[21].

For the most part the *algebraic chemistries* discussed here are of biological/-chemical nature. Thus I will refer to *algebraic chemistries* as *reaction networks*.

Closure

Definition 'A set $C \subseteq \mathcal{M}$ is *closed*, if for all reactions $(A \to B) \in \mathcal{R}$, with A a multiset of elements in $C (A \in \mathcal{P}_M(C))$, then B is also a multiset of elements in $C (B \in \mathcal{P}_M(C))$.'(Dittrich and Speroni di Fenizio, 2007, p. 1203 in [12]).

This means, a subset of $\mathcal{M}$ is called *closed* if for all reactions where all species from the reactant side of the reaction, are only producing species already in the subset (so all species from products are also in the subset).

Semi-self-maintenance

Definition 'A set of molecules $S \subseteq \mathcal{M}$ is called *semi-self-maintaining*, if all molecules $s \in S$ that are consumed within S are also produced within that set S.'(Dittrich and Speroni di Fenizio, 2007, p. 1204 in [12]).

Considering the stoichiometric coefficients of all the reactions where all partaking species are in the subset of $\mathcal{M}$. If one would subtract the amount of reactants from the products and the result is for all species in the subset equal or greater than zero, the subset is semi-self-maintaining (only one reaction needs to produce that species to be semi-self-maintaining).

Self-maintenance

Definition 'Given an algebraic chemistry $\langle \mathcal{M}\mathcal{R} \rangle$ with $m = |\mathcal{M}|$ molecules and $n = |\mathcal{R}|$ reactions, and let $S = (s_{i,j})$ be the $(m \times n)$ stoichiometric matrix implied by the reaction rules $\mathcal{R}$, where $s_{i,j}$ denotes the number of molecules of type i produced in reaction j. A set of molecules $C \subseteq \mathcal{M}$ is called self-maintaining, if there exists a flux vector $v \in \mathcal{R}_n$ such that the following three conditions apply: (1) for all reactions $v(A \to B)$ with $A \in \mathcal{P}_M(C)$ the flux $v(A \to B) > 0$; (2) for all reactions $v(A \to B)$ with $A \notin \mathcal{P}_M(C)$, $v(A \to B) = 0$; and (3) for all molecules $i \in C$, the production rate $f_i = (Sv)_i \geq 0$ with $(f_1, ..., f_m)T = Sv$. $v(A \to B)$ denotes the element of v describing the flux (i.e. rate) of reaction $A \to B$. f_i is the production rate of molecule i given flux vector v. It is practically the sum of fluxes producing i minus the fluxes consuming i.'(Dittrich and Speroni di Fenizio, 2007, p. 1205-1206 in [12])

This can be casually summarised as: if a stoichiometric matrix S (entries $S_{i,j}$ are the number of species i produced by reaction j) formed from the reactions only including the species of a subset of $\mathcal{M}$ and multiplied by a flux vector v with $v > 0$ so that $Sv \geq 0$ then that subset is called *self-maintaining*. If there exists no such vector v then the set is not *self-maintaining*.

1.2.2 Semi-Organizations

If a set $S \subseteq \mathcal{M}$ is *closed* and *semi-self-maintained* it is called a semi-organization.[12]

1.2.3 Organization

If a set $S \subseteq \mathcal{M}$ has all three properties, *closed*, *semi-self-maintained* and *self-maintained*, then it is called an organization.[12] Since an organization is also *closed* and *semi-self-maintained* one can conclude that all organizations are also semi-organizations.

1.2.4 Finding Organizations

Finding organizations is not an easy task. It is also not computationally trivial. Dittrich and Speroni di Fenizio propose a two step method for finding semi-organizations.

First all semi-organizations are computed, then these are checked for *self-maintenance*.[12] This is exactly what the *Constructive Algorithm* does. The next section will cover this in more depth.

1.3 Computation of Chemical Organizations

Determining organizations by hand is a difficult and tedious undertaking. Considering the structure of a lattice[1] and with that all the possible closed sets and semi organizations it is easy to see why calculating the closed and self-maintained sets is not an option. The number of possible sets to check in a small network with a naïve approach quickly becomes unfeasible. Trivially known from combinatorics the number of possible organizations in a network is equal to 2^n (n being the number of species in $\langle \mathcal{MR} \rangle$).

OrgTools

In 2008 Centler *et al.*[7] presented three algorithms to compute chemical organizations given by a reaction network in sbml or rea format. Two of these algorithms, the Constructive Algorithm and the Flux Based Algorithm, compute all organizations, while the third algorithm uses runtime heuristics and thus does not guarantee the uncovering of all organizations, but a substantial amount of organizations will be found.

[1] Algebraic concept, see Wikiedia[52]

Constructive Algorithm

The Constructive Algorithm presented by Centler *et al.*[7] works in two steps. First all semi-organizations of $\langle \mathcal{MR} \rangle$ are determined. In a second step all these semi-organization are tested for self-maintenance. This two-step method was also proposed by Dittrich and Speroni di Fenizio in 2007[12].

Computing all semi-organizations is done in a bottom-up approach. Starting with the empty set, the closure of it is computed. Adding the closure to an array, which will hold working sets, the loop for iteratively computing all semi-organizations has begun. While this array holds a semi-organization, the algorithm will search for all semi-organizations directly above the working set. Thus found semi-organizations will be added to the array. With *semi-organizations directly above* are meant the sets with a cardinality one bigger than the working set. Remembering the lattice from before, these are the sets *above* the working set. The current working set is then returned as a result set and removed from the array. If the array is empty and no more semi-organizations are found, the algorithm will proceed to step two.

Every semi-organization is then **checked for self-maintenance**. As mentioned above, if there exists a flux vector $\boldsymbol{v}$ such that $\mathbf{S}\boldsymbol{v} \geq \mathbf{0}$, then the semi-organization is self-maintaining thus also an organization. This can be mapped to a linear programming problem which can be solved using a flavor of the simplex method based on the work presented by Dantzig in 1963[*]. 'Since only the existence of such a flux vector v is of concern, only the first phase of the simplex method needs to be performed.'(Centler *et al.*, 2008, p. 1613 in [7]). All sets for which the simplex method could find a basis, are self-maintaining and returned as the set of organizations of $\langle MR \rangle$.

OrgAnalysis, OrgFinder and OrgView

OrgTools is a software package that includes 3 main applications.

OrgAnalysis is a UI-application which finds and displays a list view of the organizations. It also lets you export the reaction network to rea and sbml and the lattice to ltc and orgml.

OrgFinder is a command line tool for finding the organizations of a reaction network given in rea or sbml. It exports the organizations as a lattice file in rea or orgml format.

OrgView is a lattice-viewer. It displays the lattice file on screen and lets the user manipulate the horizontal position of each node (a node represents an organization).

2. Methods

Development for the web is not a completely different thing compared to developing standalone software or writing small Python scripts. However, there are some characteristics that need explanation and some choices in software or language that need elaboration. The following environments, packages, languages and libraries were used to create the web application *WebOrg*.

2.1 Node.js

Node.js (*Node*) is a JavaScript runtime environment[33]. It is cross-platform and open-source. *Node* is built upon *Google's V8 JavaScript* and WebAssembly engine. *V8* – and therefore *Node* – can run standalone[45], no web browser is needed to run JavaScript or WebAssembly[1].

However the aim is to produce a web application. In this case *Node* will be used to run a local web server whilst implementing the software. It would then be possible to access the web application from a browser inside the same network as the machine running the *Node* server.

The *Node Package Manager* (*npm*) is another point in favor of using *Node* and is one of the largest software registries[35]. *npm* is accessible through its website and a command line interface (CLI). It allows developers to 'share and borrow'[35] JavaScript packages. Packages could be JavaScript libraries (e.g., *Cytoscape.js*, see below) or standalone tools (e.g., *Create-React-App*[2]). As well as managing dependencies and package versions, *npm* also points out potential version conflicts and other problems concerning the used packages.

2.2 Typescript

TypeScript is a superset of JavaScript. It is developed by Microsoft. 'TypeScript adds additional syntax to JavaScript[...]'[44]. The main novelty – and its namesake – is the introduction of types. Code written in TypeScript is compiled to JavaScript, therefore *TypeScript* runs in the browser the same way JavaScript does.

'TypeScript becomes JavaScript via the delete key.'[44]

[1] see **WebAssembly for Computing Semi-Organizations** for possible application
[2] see **Implementation**

```
function sum(a: number, b:
    number): number {
     return a+b;
}
```

(a) `sum` function in TypeScript. Including *types* `number`.

```
function sum(a, b) {

    return a+b;
}
```

(b) `sum` function converted to JavaScript through deletion of types.

Figure 2.1: How TypeScript is converted to JavaScript for a simple case.

Conversion between TypeScript and JavaScript will not be as easy as shown a lot of the times. However, the example represents the idea behind TypeScript very well.

Until this point all libraries and packages were explained with JavaScript in mind, most of them are also distributed as JavaScript packages. The resulting web application runs in the browser with HTML, CSS and JavaScript. However, TypeScript was used to implement it. This is possible because TypeScript allows JavaScript to be used in TypeScript. When the author of a package also distributes a `*.d.ts` file, TypeScript can also check in the JavaScript files for types, essentially handling them as normal TypeScript files.

```
function sum(a, b) {
    return a+b;
}
```

(a) `index.js` file in JavaScript.

```
function sum(a: number, b:
    number): number;
```

(b) `index.d.ts` file for the `index.js` file.

Figure 2.2: An `index.d.ts` type file for an JavaScript file.

2.3 React.js

'React[.js] is a declarative, efficient, and flexible JavaScript library for building user interfaces'[39].

As this minimal definition states *React.js* (*React*) is a JavaScript for the creation of user interfaces (UI). The library is developed by Facebook Inc. The declarative nature of the way elements are defined in the UI is the main reason it was chosen over plain JavaScript. In *React* the actual HTML part is negligible. UI components are created in JavaScript and exported as *React* components. This allows for dynamical loading and deleting of component. For example, when displaying the list of organizations, it would be enough to define a list component which holds all desired information for one element of the organization list and then instantiating one of these list components for each of these organization elements and populating it with data.

React components are Classes. Custom components can be created if the basic Class component is extended. The custom component needs to implement a `render`

function. All other functions are implemented in the basic *React* component but can be overridden. A component can receive a `props` object. These Properties (Props) can for example include data needed to change some Dropdown component from being collapsed or expanded.

A Class is not always desired or the other functions implemented in a standard component are fine the way they are. It is possible to extend a *React* Function component. Essentially this allows a developer to define only the `render` function and call it the same way you would call a complete Class component. Props are passed the same way as in a Class component.

React also has the ability to work with additional packages. Two of these are described below (*Redux* and *MUI*)

2.3.1 Create-React-App

Create-React-App lets one create a minimal *React*-app with just one command. It has support for templates, such as a typescript template, which was used for this web application. *Create-React-App* sets up Webpack under the hood and hooks all the needed scripts and configurations up, so that there is only one more command left to be used before seeing a *React*-TypeScript-App running in one's browser.[14]

Webpack *Webpack* is a bundler for JavaScript using a JSON[3] configuration file to specify rules before bundling. It combines multiple JavaScript files to one, this includes external static bundles. All packages only including what is needed, resulting in a smaller overall build size. It also triggers the TypeScript compilation and provides *Hot Module Replacement*[4] during development.[51]

2.3.2 Material UI/MUI

Material UI (renamed 2021 to *MUI* and thus used in this work) is a UI library for *React*[25].
It provides *React* component definitions for common UI elements following the design guidelines of Google's *Material Design* (hence the naming). It also provides an extensive style API, so all predefined components can be customised. The style API also offers the ability to animate components. Should the *Material Design* not be desired, *MUI* allows for custom theming.

The code above is an example implementation of a *React* Functional component including predefined *MUI* components (here `ListItem`, `ListItemText`). In this example the Props passed to `OrganizationListElement` are `org` with type `Org` (an Organization Class). To demonstrate that variables can also be defined in a functional component, the constant `organization` is set to the property `org`. `return` will provide the definition of the *React* Function component, therefore returning what will be displayed in the browser.

[3] **J***ava***S***cript* **O***bject* **N***otation* is a language independent data-interchange format.

[4] Allows for changes to the code while the bundled website is running. After saving *Webpack* rebuilds automatically and updates the website at runtime.

```
function OrganizationListElement(props: {org: Org}): React.FC {
    const organization = props.org;
    return (
        <ListItem button onClick={()=>{ /* do something */}}>
            <ListItemText primary={`Organization: ${organization.id
                }`} sx={{ color: '#002350'}}/>
        </ListItem>
    );
}
```

Figure 2.3: Example of a *React* List Element component implemented in TypeScript with *MUI*.

As mentioned `ListItem` and `ListItemText` are *MUI* components. All *MUI* components have an API. Not all of them have the same properties available. However, the `sx` property is available for most *MUI* components. It handles simple CSS commands. In this example the color of `primary` was changed. This will change the primary text[5] of the `ListItemText` component to a darker shade of blue. The `button` property of `ListItem` indicates that the `ListItem` component should be a button and should therefore behave like one. There should be a hover effect when the mouse enters the component on screen and there should be a visible effect when the button is pressed and released. All these effects are handled by *MUI*, but – as specified above – can be customised using the theming API. The `onClick` event can be set directly in the component or a function could be defined outside of the `return` to be executed after a click on the `ListItem`.

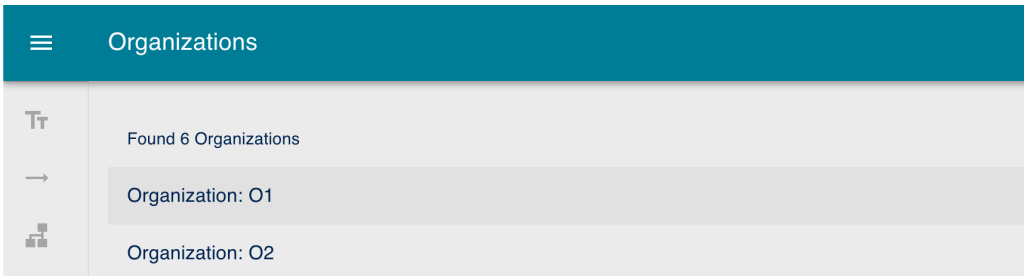


Figure 2.4: Two `ListItems` from Figure 2.3. *O1* with hover effect

2.4 Redux

Handling data on a website can be complicated. Is a user logged in, what is in the users shopping cart or – in this case – were organizations found and which elements

[5]For `ListItemText` there is also a `secondary` property which would display a text below the primary text in a smaller font size. All available components and properties are documented at *MUI* documentation page.

are included? *Redux* is a state container for JavaScript[3]. A *Redux* State is the current slice of a *Redux* Store. The Store can be modified by calling an actions (function like behaviors). These actions will not modify the Store directly, they will return a modified copy – a slice – of the Store resulting in a new State. When using *Redux* one should never modify the Store but treat it as immutable[1]. This helps in predictability and robustness of the Store thus the application. *Redux* also offers official bindings for *React*.[2] These will help with integrating a *Redux* Store in a *React* application. They will also trigger re-renders for changes in the *Redux* Store.

2.5 WebAssembly

WebAssembly aims to be a new web standard.[46] It compiles a host of languages to binary instruction format[6]. 'WebAssembly aims to execute at native speed by taking advantage of common hardware capabilities available on a wide range of platforms.'[49] There are however limitations. For example, AssemblyScript is a TypeScript-like language which is compiled to binary instruction format via the *AssemblyScript Compiler*[7]. Although handling numbers is fast, handling strings and complex objects requires more time.[42] In this project WebAssembly is used to make a C/C++ library[8] run on the web and be callable from TypeScript code.

2.6 glpk.js

Moving from the preceding section to the package that utilises WebAssembly. *glpk.js* is a JavaScript package – written in TypeScript – creating a JSON interface for *GLPK – GNU Linear Programming Kit –*, a linear programming solver from the GNU Project written in ANSI C. With *emscripten* the *GLPK* library is compiled to the WebAssembly binary instruction format. A JavaScript file works as an interface between WebAssembly and the TypeScript/JavaScript application. First a LP-Object[9] is defined; then the *glpk.js*-Interface calls a `solve` function, which takes the LP-Object and – optionally – an options object[10] and sets everything up for the WebAssembly package to work. This includes converting the LP-Object via the GLPK commands (e.g. `glp_create_prob`, `glp_add_rows` and `glp_load_matrix`[10]) to a *LP-Object* that the *GLPK* WebAssembly – therefore the C version– can understand. `solve` tries to compute the LP-Object with the *simplex algorithm*[11]. `solve` returns a Result-Object containing a status and the result of the *simplex algorithm*[7].

[6]Languages used include: C/C++, Rust, AssemblyScript, C#, Go, Swift and more.[47]

[7]Other languages have other compilers. One of the more established compilers is *Emscripten*[13] for C/C++

[8]see **Implementation**

[9]see: **Implementation** for more information on this topic.

[10]Including for example the messaging level (how much feedback is printed to the console), pre-solving, callback function (function that is executed after `solve` is done).

2.7 cytoscape.js

With *glpk.js* it is possible to check if a flux vector exists. Following the definitions from Chapter One the set of organizations is now known. The definitions also state that these organizations form a lattice structure. To visualise this lattice structure the open-source JavaScript-based library *cytoscape.js*[19] is used. Important features for this project are the ability to display custom styles for nodes and edges, also being able to have different styles for an element type in the same graph, manipulating the graph in runtime, applying layouts and exporting the graph as an image. It is also possible to extend the library through extensions, two extensions will be discussed in **Implementation**.
A *cytoscape.js* graph is stored in a JSON data structure with node and edge elements. A node element is also a JSON object at least containing a `data` object. The `data` object includes an optional string element `id`. An edge element is a JSON object too. It contains at least a `source` and `target` element. These correspond to the node's `id` stored in `data`.

2.7.1 cytoscape-dagre

cytoscape-dagre is an extension for *cytoscape.js*. It is a layouter using the Directed acyclic graph system.[17]

2.7.2 cytoscape-automove

This extension for *cytoscape.js* handles node positions.[18]

2.7.3 cytoscape-svg

cytoscape-svg extends the ability of *cytoscape.js* from exporting raster images (JPEG/PNG) to vector graphics.[23]

2.8 Smaller Packages Used

Most of the following packages were used for convenience.

Monaco Editor The *Monaco Editor* is a code editor for the web written in TypeScript and developed by Microsoft. It is based on the source of Visual Studio Code with some changes to make it work in a browser.[26] It has support for custom syntax highlighting[27] which was utilised to create a rea-highlighter.

react-dropzone *react-dropzone* is a package that provides a simple *React* component to create drag and drop zones on react apps.[38]

comlink The *Comlink* package helps communicating with a web worker. It also has the functionality to send callback functions to a worker, which is important for the implementation of the constructive algorithm.[41]

react-app-rewired As mentioned above, a *create-react-app* application uses *Webpack* under the hood for user-friendliness. However, since the compiled TypeScript files will be combined to a single JavaScript file if nothing is changed in the configuration, the *Webpack* configuration file needs to be changed. *react-app-rewired* makes this possible.[5]

worker-loader Now that the *Webpack* configuration can be changed, workers need to be specified. A web worker will be called by specifying the script which needs to be run to the worker api.[31] The *worker-loader* package configures *Webpack* to save the worker scripts in a separate file.[50]

file-dialog *file-dialog* lets one call the file browser from code and triggers an upload.[34]

FileSaver.js *FileSaver.js* provides a simple function for saving a file on the clients machine (triggering a download).[20]

xml-js *xml-js* converts XML to JSON and back again. It keeps the order of elements from the XML file in the JSON which is important for reading and writing correct indices to species.[11][53]

[11]To get a feeling why this is important, see **Input/Output** to understand the connections between SBML and OrgML/ltc

3. Results

3.1 Design and Implementation

This section will guide the reader through the core components that make up WebOrg. The aim of this section is not to provide a full step by step tutorial on how to rebuild WebOrg, nor any other web application, but rather an explanation of how WebOrg is constructed.

The following will not cover the implementation in order, but is sorted so that it is comprehensible.

3.1.1 React App

The easiest way to start a new *React* application is through *create-react-app*. This will, as stated above, populate a minimum setup for a *React* application ready to be run.[1]
The entry point for the website on a web server will be the `./public/index.html`.[2] A script check and a `<div>` are placed in the `<body>`. The `<div>` has an `id` set to `"root"` which will indicate the entry point for *React* and its components. Every component added via the *React* TypeScripts below are placed inside this `<div>` component.[3]

To get *React* to know that components are declared a small setup file is necessary. `./src/index.tsx` will take this role. The first thing that stands out is its file ending. TypeScript usually starts with a `.ts` ending. `.tsx` (or `.jsx` in JavaScript) indicate to *Webpack* that this file is a *React* component, or needs interpretation in a *React* context.[15] TSX/JSX is a syntax extension to allow for TypeScript/JavaScript in HTML templates. Figure 3.1 shows a small example of the way TSX can be used. The curly brackets allow for any TypeScript operation inside.
The function `ReactDOM.render()` will provide a *React* component (first argument) to the element passed as second argument. The provided component in this case is `<Main />` which is the WebOrg main component.[4]

The `./src/Main.tsx` file contains the definition for the `<Main />` component.

[1] the root of this project will be the current directory (`.`) in any paths given in this Chapter

[2] See `./public/index.html` in the **Appendix 5.1**.

[3] A web developer might note, that a `<script>` component is missing for a TypeScript/-JavaScript component to be run. That is true and this will be added by the *Webpack* build process. See `./build/index.html` in the **Appendix 5.2** to compare.

[4] See `./src/index.tsx` in the **Appendix 5.3**.

```tsx
function Greeting: React.FC () {
    const name: string = "Bob";
    return (
        <h1>Hello, {name}!</h1>
    );
}
// Expected: Hello, Bob! to appear on the website.
```

Figure 3.1: Example TSX usage in a *React* component.

Material UI

The `./src/Main.tsx` consist mostly of UI definitions for screen elements[5]. The screen space is always occupied by at least three elements. All three are realised using the *MUI* library and the components it provides.

Defining the AppBar component *MUI* provides an `AppBar` component, which was extended to accept an `open: boolean` property and allows for animation when opening the `Drawer` component.
The `AppBar` component contains a `<MenuIcon />` provided by *MUI*'s vast Icon Library.[24] The icon is only visible when `open === false`[6] Similarly will the name

```tsx
// ...
<IconButton sx={{display: {open ? "none" : "block"}}}>
    <MenuIcon />
</IconButton>
// ...
```

Figure 3.2: Example of how `open` determines the visibility of the icon.

of the active tab switch and if the name is equal to `"Lattice"` the `<NodeViewDropdown />`[7] component is shown.

```tsx
// ...
<Typography>{title}</Typography>
{title==="Lattice" ? <NodeViewDropdown /> : <div />}
// ...
```

Figure 3.3: Example of how `title` determines the visibility of the dropdown component and the text displayed in the `AppBar`.

On the right end the `AppBar` hosts the always visible `Settings` icon.

[5] Elements displayed on screen.
[6] JavaScript/TypeScript characteristic. `===` is a *strict equality* operator. `A===B` checks if `A` and `B` are equal. See **Appendix 5.1** for more information.
[7] See **Appendix 5.5** for source code.

Animated Drawer The side-bar component is called a `Drawer`, if it can be slid out and in. In this web application – following the documentation closely – this is done via a custom theme, that specifies the animation.[8] The Drawer contains all the `ListItems` which are used to navigate the application. The `ListItems` include four main items which all have their own collapsable items. Figure 3.4

```
// ...
<ListItem button onClick={()=>{handleClick('Editor')}}>
    {listIcon('Editor', <TextFieldsIcon />)}
    {listLabel('Editor')}
    {openE ? <ExpandLess sx={{color: primary}}/> : <ExpandMore sx
       ={{color: secondary}}/>}
</ListItem>
<Collapse in={openE} timeout="auto" unmountOnExit>
    // ...
</Collapse>
// ...
```

Figure 3.4: Example of how the `ListItems` are implemented.

shows how the `ListItems` are implemented. Each item contains the output of two functions and the output of a check whether the item is expanded or not. The two functions – `listIcon()` and `listLabel()` return a `ListItemIcon` or a `ListItemText` component respectively. Functions are used here to change the color from a darker blue to a gray when not active (expanded).

Tabs and TabPanel To quickly switch between screens without reloading the application a *Tab*-system is used. Each of the four main screens – Editor, Reaction Network View, Lattice View and Organization List – have their own tab. These are all loaded upfront[9] and can then be used as needed without download times inbetween. Figure 3.5 shows how the `TabPanel` component is used. This component is not directly a *MUI* component, but a custom one following the instructions of the *MUI* documentation.[10]

3.1.2 Custom Components

Not all components can be generated using *MUIs* component library. These were then realised by creating a number of custom *React* components.

Editor

The *Editor* component is composed of two parts. A *react-dropzone* component and a *Monaco Editor* component. Both packages provide a *React* component. However

[8] See **Appendix 5.6** for source code.

[9] 'loaded upfront' meaning all JavaScript and other needed components are loaded, calculations or layoutings need to be done when selected.

[10] See the **Appendix 5.7** for the source code of the `TabPanel` component.

```
<TabPanel value={title} index={'Editor'}>
    <Dropzone />
</TabPanel>
<TabPanel value={title} index={'Reaction Network'}>
    <ReactionNetwork ref={networkRef}/>
</TabPanel>
<TabPanel value={title} index={'Lattice'}>
    <LatticeGraph ref={latticeRef}/>
</TabPanel>
<TabPanel value={title} index={'Organizations'} >
    <OrganizationList />
</TabPanel>
```

Figure 3.5: Example of how the `TabPanel` is used.

not only can *react-dropzone* be used as a component, but it also provides state bindings through the `useDropzone()` function. These can then be passed to other

```
<div {...getRootProps()} style={{
    height: '100%'
}}>
        <input {...getInputProps()} />
    <Editor
        width='100%'
        height='100%'
        theme={`org-theme-${currentType}`}
        options={options}
        defaultLanguage={currentType}
        onChange={handleEditorChange}
        beforeMount={editorWillMount}
        value={text}
        key={currentType}
    />
</div>
```

Figure 3.6: Encapsulation of the *Monaco* `Editor` component by *react-dropzone* components.

components, giving them the functionality of the *react-dropzone* component. This can be seen in Figure 3.6.
The `rootProps` component encloses then an `input` component and the *Monaco* `Editor` component. The `Editor` gets passed a theme, options and a language type (for syntax highlighting).[11] The code shown in Figure 3.6 will be returned to *React*, but only the component displayed will be the `Editor`.

[11] See **Appendix 5.8** for the *Monaco* `Editor` arguments.

Cytoscape Components

Both – the *Reaction Network* and the *Lattice* – use *cytoscape.js* to display graphs on the screen. Following the example of Kenneth Pirman from his CodeSandbox repository, *cytoscape.js* is integrated into a *React* component.[36] Both components define a layout option and get stylesheets declared for them.
One uses the *React* function `useImperativeHandler()` to expose methods in a parent component. The methods exposed are `draw()` and `takeImage()` in both graph-components. `draw()` (re-)draws the current graph on screen. `takeImage()` calculates a *svg* and triggers a download.

Organizations List

The *Organization List* component maps a `ListItem` to a `List` component for each found organization. Using the same technique as used in the `Drawer`, the `ListItem` is populated with a dropdown. The expanded components show the included *Species* and *Reactions*.
On top of the list sits a `ListSubheader` containing the number of found *Orgsnizations*.

3.1.3 Classes, Types and Interfaces

Species

The *Species* interface represents a collection of things, that each species must include when read into WebOrg.
A Species needs an `index`, which is a unique identifier, because it is a continuous count while importing species. The `uid` is a unique string identifier corresponding to the 'id' in a `sbml`-species. Also an `id`, which corresponds to a 'name' and an optional `meta` object, constructed of key-value pairs.
It is exported for use as a `type`.

```
interface Species {
    uid: string,
    id: string,
    index: number,
    meta: {[key:string]: any}
}

export type { Species };
```

Figure 3.7: Implementation of the *Species* interface.

Even though it looks very simple and not worth the effort, defining a type assures that all needed attributes are set for an object of type *Species*.

MultiSet

Unlike *Species* *MultiSet* is a class. The class contains a `hashString` (string), the `cardinality` (number), a `species` (array) and a `booleanArray` (array). Additionally a `constructor` – taking a `species` array and the number of distinct *Species* of the complete network – and multiple getters and setters[12].

The `hashString` is a string with a length of the number of *Species*. Each position in the string represents one *Species*. Is a *Species* contained in the `species` array, then the position corresponding to `species.index` is set to `"1"` otherwise it is set to `"0"`. This allows for an easy way of comparing two sets. If the `hashStrings` are equal, the *MultiSets* are also equal.
It also is used to set an entry in the *Hash*, see below.

The *cardinality* variable represents the cardinality of the set of *Species* represented by the *MultiSet*.

The `species` array represents the set of *Species*.

For fast checks if a single *Species* is contained in the *MultiSet* the `booleanArray` is similar to the `hashString`. Each element in the array represents a *Species*. Is the *Species* contained in the set, then the element is set to `true`, if not it is set to `false`.[13].

Reactions

Reactions are similar to *Species* also defined by a simple interface. The *Reaction* interface consists of an `index`, analog to `Species.index`, an `uid` string – built from the components of the reaction –, a boolean variable `reversable`, two *MultiSets* – one for reactants and one for products –, and the `stoichiometry` array.
Again exported as a `type` for later use. The `uid` is mainly for debugging purposes.

```
interface Reactions {
    index: number,
    uid: string,
    reversable: boolean,
    reactants: MultiSet,
    products: MultiSet,
    stoichiometry: number[]
}

export type { Reaction };
```

Figure 3.8: Implementation of the *Reaction* interface.

For a human readable string, each species of reactants and products are added – separated by `->` – to the `uid` string with the corresponding stoichiometric coefficient in brackets. Resulting in an `uid` like so: `A[1] B[1] -> A[1] B[2]`

[12] functions to get and set properties of a class
[13] The source code can be found in the **Appendix 5.9**

Hash

The *Hash* class extends on a simple `Map<string, boolean>`[14] array. This allows to check if a string is already inside of the `Map` or not. If a string is inside, the *MultiSet* to which the string belongs was already processed. If it is not, the hash adds it to the `Map`.
The class makes three functions available. `freeHash` clears all entries of the *Hash*, `freeHashBelow` clears all entries below a number given and `insert` tries to insert a `hashString` from a *MultiSet* into the *Hash*. The `insert` function returns `true` if the *MultiSet* could be inserted, `false` if not.

This *Hash* class is based on the Hash class from Centler *et al.*, 2008.[7][15]

Organization

The *Organization* interface defines an object with an `id`, an array of *Species*, an array of *Reactions* and a two-dimensional array `stoichiometry`.

```
import { Reaction, Species } from "../";

export interface Organization {
    id: string;
    species: Species[];
    reactions: Reaction[];
    stoichiometry: number[][];
}
```

Figure 3.9: Implementation of the *Organization* interface.

The `stoichiometry` array is equivalent with a $n \times m$ matrix – where n is the length of the `reactions` array and m the number of unique *Species* in the whole network. All *Reactions* contained in the `reactions` array are made up of only *Species* from that *Organization*

3.1.4 State Management

As previously mentioned *Redux* is used to set up a state management system. For that purpose *Redux* uses an object called `store`. As Figure 3.10 shows, a `store` consists of `reducers` and `middleware`. Since the `middleware` is not changed in any major way, it will just be noted that time intensive checks were disabled. Using TypeScript it is not necessary to check if the type is serialisable, because it is known which type is passed to the `reducer`. A `reducer` is a function that will be used to create a new `state`. It has to have the following structure: `(oldState, action) => newState`.[4] The two `reducer` objects are discussed below.

[14] A `Map` is similar to a dictionary in other languages.
[15] The source code can also be found in the **Appendix 5.10**

```
export const store = configureStore({
  reducer: {
    app_reducer: appReducer,
    network_reducer: networkReducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
        serializableCheck: false,
        immutableCheck: false,
    })
});
```

Figure 3.10: Implementation of the *store* object.

AppState

The *AppState*-`reducer` object (*AppState*) hosts all elements that effect the web application itself and need to be stored when anything would change the element referencing it. This includes the current text displayed in the *Editor*, a boolean element to check if the current text has been read or not, if the verbose mode is active, if a thread is running, what the current active tab is, what the current file type in the editor is, the current network graph, the current lattice graph, the current list of organizations (not the same as the organizations stored in the *NetworkState*, here for user interface) and the current node label type.

All these stored objects need a `reducer` to change them in a later state. A sample

```
toggleVerbose: (state, action: PayloadAction<boolean>) => {
    state.verbose = action.payload;
},
```

Figure 3.11: Sample `reducer` for toggling the *verbose mode*.

reducer is shown in Figure 3.11. Calling this `toggleVerbose()` reducer in another component is called dispatching. This is because the function called from the *Redux* `store` used to pass a function call to the current state is called dispatch.

NetworkState

The *NetworkState* contains objects for all *Species*, all *Reactions*, all found semi-organizations (as *Organization* objects) and all found organizations (also as *Organization*) in the reaction network. It only contains information directly concerning the reaction network.

This split is not necessary but increases readability of the `store`.

3.1.5 Threading

Actual multi threading was not implemented, however moving the heavy computations out of the main thread frees the webpage up to be responsive. This was

proposed to become the new best practice for web development by Surma.[16][40] This was realised with the *comlink* package. Shown is the setup and start of a worker-thread using *comlink* for the computation of all semi-organizations.

```
const worker: Comlink.Remote<OrganizationWorker> = Comlink.wrap(new
    organizationAPI());
worker.calculateOrganizations(connected, verbose, species,
   reactionList, Comlink.proxy(callback));
```

Figure 3.12: Example of how *comlink* handles the start of a worker-thread. For the implementation of the `OrganizationWorker` interface see Appendix 5.11.

3.1.6 The Constructive Algorithm

Earlier I elaborated how the *Constructive Algorithm* works. This section will evaluate the TypeScript implementation. It will not cover the whole implementation, but describes the fundamental and important parts.

Computing Semi-Organizations

As the section earlier in this work covered, the *Constructive Algorithm* works in two parts.
First all semi-organizations need to be computed. This works mainly the same way as the implementation provided by Centler *et al.* with some peculiarities due to the introduction of the *MultiSet* class and changes in TypeScript/JavaScript syntax.

Starting by calculating the closure of the empty set and then calculating semi-organizations above as long as there are more semi-organizations to work from going further up the *Lattice*.

This will result in an array of numbers, representing the semi-organizations found. The length of the array is $l = n * m$, with n species and m found semi-organizations.

Testing for Self-Maintenance

After all semi-organizations are found, all that is left to do is to check whether or not they are self-maintaining. The semi-organization array is sliced into array of the length n(see above). From that smaller array an organization is created. This happens through the formulation of an LP-problem and testing it via the simplex-method. This is done through the WebAssembly bindings of the glpk library.

The formulation of the LP-problem is done in two steps. First one searches for all *Reactions* involved in this semi-organization. This includes reactions that only contain *Species* that are also included in the semi-organization. If none are found, then the semi-organization is also self-maintained, since any vector $\vec{v}$ would be a suitable solution to $S\vec{v} \geq \vec{0}$.

[16]Surma is a Web Advocate at Google and author of the comlink package.

If *Reactions* are found, the formulation proceeds to step two. In step two the LP-problem is defined. First by creating an empty problem, then populating the objective function with a variable for each *Reaction* and adding a constraint for each *Species*. The constraints are generated from the stoichiometry of the *Species* in the *Reactions*. Since only phase one of the simplex-method is needed, the objective coefficients are set to zero.[17][6] Figure shows a LP-problem defined for the semi-organization `{A,B,C}` from the *simpleNetwork.rea* file, provided by Dittrich and Speroni di Fenizio[12].
If the simplex returns a solution an updated *Organization* object is returned.

The algorithm concludes by dispatching all semi-organizations and organizations to the `store` and triggering the generation of the *Organization List*.

3.1.7 Generating Graphs

Graph generation is used in two parts of OrgWeb, generating a *Reaction Network* from a file and forming a *Lattice* from found organizations.

Reaction Network Generation

The *Reaction Network* graph is generated from the list of *Species* and *Reactions*.

First every *Species* and every *Reaction* get a node added to the list of nodes needed by *cytoscape.js*. How a node is defined is shown in Figure 3.13. Nodes get a

```
// NodeDefinition for Species
const node: NodeDefinition = {
    data: {id: species[i], width: 50, height: 20, color: '#000',
       level:  4},
};

// NodeDefinition for Reactions
const node: NodeDefinition = {
    data: {id: `${i}`, label: 'reaction', width: 50, height: 20,
       color: '#000', level: level}
};
```

Figure 3.13: Sample node generation for *Reaction Network*.

level mapped to them. The level is used to place a node on a ring of the *concentric-layout*. This layout places nodes on a given number of circles. In this case there are three circles nodes can be added to. The smallest circle, corresponding to level 5, is the circle for normal *Reaction* nodes. The circle in the middle, corresponding to level 4, is the circle to where the *Species* nodes are added. The outer most circle contains special *Reaction* nodes. These are in- and outflow *Reactions*. They have the level 0.

Each *Reaction* is then looped again and an edge is drawn between reactant *Species* node and *Reaction* node. If the *Reaction* node is a level 0 node the edge

[17] This is so that any feasible solution is the optimal solution to the objective function.

```
// EdgeDefinition
const edge: EdgeDefinition = {data: {id: `R${reaction[k]}${i}${iter
    ++}`, label: `${reaction[k] !== 1 ? reaction[k] : ''}`, source:
    '', target: ''}};

if(k < species.length) {
     edge.data.source = species[k];
     edge.data.target = `${i/(species.length*2)}`;
     edge.data['educts'] = true;
     if(reaction.slice(species.length).every(item => item === 0)) {
         edge.data['outflow'] = true;
     }
     if(reaction[k] > 0) {
         this.edges.push(edge);
     }
} else {
     edge.data.source = `${i/(species.length*2)}`;
     edge.data.target = species[k-species.length];
     edge.data['products'] = true;
     if(reaction[k] > 0) {
         this.edges.push(edge);
     }
}
```

Figure 3.14: Sample edge generation for *Reaction Network*.

gets an arrow-tip at its *Reaction* end. Also from the *Reaction* node to every product *Species* node is an edge with arrow-tip drawn. Figure shows how this is defined.

After both are generated, the results are dispatched to the `store`. After the store changed, a redraw of the graph is triggered and the *Reaction Network* will be displayed on screen.

Lattice Generation

Graph generation for the *Lattice* works a little different. First the nodes are placed. Each node corresponds to an organization. Figure 3.15 shows the node definition for the organization. Each nodes vertical position is directly dependent on the cardinality of the organization. The horizontal position is done by the dagre layouting[17].

```
// NodeDefinition for Species
const orgObject = {
     data: {id: `O${index}`, label: `O${index}: ${vert}`,
         normalWidth: 50, all_species: all_species.substring(0,
         all_species.length-2), all_species_width: allspecieswidth,
         novelty_species: '', novelty_species_width: 20, rank: vert,
         height: 20, color: '#000', },
};
```

Figure 3.15: Sample node generation for *Lattice*. Saving every label and width directly for the three possibilities of displaying the label.

The edges are calculated in a trickle down manner after the nodes are all generated. This is done by searching for nodes below the current node, the current node could be generated by adding new *Species*. First one searches directly below then when not all *Species* could be produced one goes even lower. If the lowest node is reached and a *Species* could not be reached, it will be added to the *Novelty Species* label.

As done in the generation of the *Reaction Network* generation, after all edges are generated, everything will be dispatched to the `lattice_graph` in the `store`. The change will then trigger the redrawing of the *Lattice* and it will be displayed on screen.

3.2 User Experience

3.2.1 User Interface

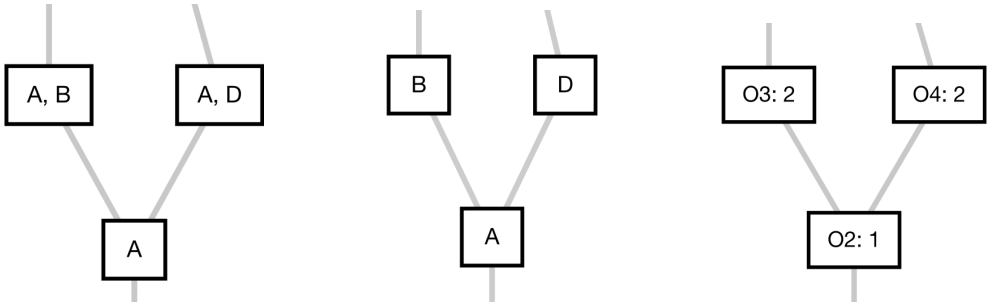
The user interface (UI) is the users first impression of the application. Creating a UI is a balancing act between having everything accessible as quickly as possible and not overwhelming the user by having buttons and forms everywhere. An optional target was to look 'clean and modern'. At the time of writing *MUI* or the *Material Design* in general are considered 'modern'- and – subjectively – 'clean'-looking, the main challenge was to have every option accessible with the least amount of clicks possible, while also being organised.

AppBar

The AppBar contains mainly the information about which tab is currently open. Figure 3.16 shows the AppBar. It contains (from left to right) a side-bar icon (indicating that the side-bar is closed), the active tab label (in this case the *Lattice tab*), the *Node Label* selector and the settings button.

Figure 3.16: The AppBar with the *Lattice tab* active.

The *Node Label* selector is only visible if the active tab is the *Lattic tab*. This ensures that it is visible when its needed and not distracting when not. Nothing would change if the *Node Label* is changed in the *editor tab* for example. There are three options to choose from the *Node Label* selector: First *Species*, this will display all elements contained in the organization inside the node. Second *Novelty species* will show only new elements not previously contained in one of the organizations directly below. Last is *Org index*. This displays the organization index and its cardinality. Figure 3.17 shows the differences between the three selections.

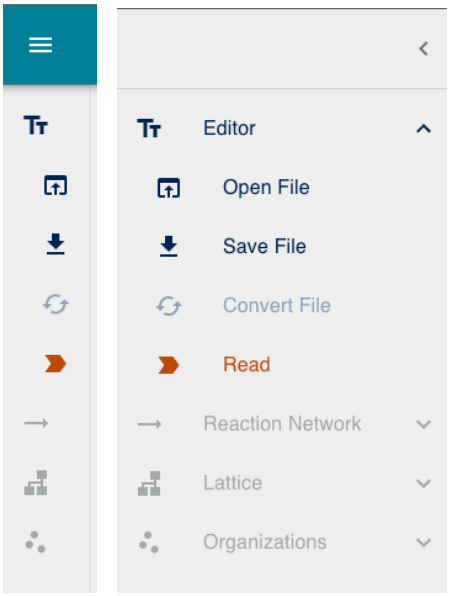


(a) *Species.* (b) *Novelty species.* (c) *Org index.*

Figure 3.17: The difference in *Node Label* for the different selections.

Side-bar

The side-bar is where the user accesses the main functionality. There are four main buttons, each containing an icon and a text label. The text label is only visible in an open state side-bar. These four buttons correspond to a tab. They are (from top to bottom) buttons for the *Editor* tab, the *Reaction Network* tab, the *Lattice* tab and the organization tab. If a tab is active (in Figure 3.18 the *Editor* tab) it is coloured in a darker blue and a collapsable dropdown expanded. Each tab has its own dropdown, with different elements. As Figure 3.18 shows the *Editor* tab is active and its dropdown is expanded. The dropdown contains four more buttons. The two top most buttons are for input and output (I/O). The *Open File* button will open an open file dialog (which is operating system specific) and will allow for file of type **SBML** and **REA** to be opened. The *Save File* button will pack the current text inside the *Editor* and start downloading it as *Network.xml* or as *Network.rea*. Just below is an always grayed out button – indicating it is disabled – this is a placeholder button for a conversion functionality between **SBML** and **REA** or vice versa. The bottom button is the *Read* button. It has two states and two colours matching the state. If the text inside the *Editor* differs to the text in the *Redux* store, the text must first be read before new organizations can be computed. This is indicated by an orange colour. Is the text identical to the one in the store the colour is changed to the same shade of blue as the other active buttons. Clicking this button will trigger the dispatch of a new state with an updated text.



(a) (b)

Figure 3.18: (a) shows the closed, (b) the open side-bar

The other three expanded dropdowns are shown in Figure 3.19. The buttons

should all be self explanatory. The *Draw* buttons trigger a draw event. The graph is computed, the *cytoscape.js* draws the graph to screen.

(a) *Reaction Network* (b) *Lattice* (c) *Organizations*

Figure 3.19: Different expanded dropdowns

Save Image buttons activate the collection of graph data, which is then converted to a scalable vector graphics (**svg**)[37] and then downloaded. This enables scaling the image without a resolution loss. This is very useful especially with big networks or lattices. The two buttons – *Save Lattice* and *Save Organizations* – trigger – as *Save File* – a text file download. *Save Lattice* will download a **lattice** file and *Save Organizations* an **OrgML** file.

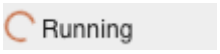


Figure 3.20: Indicating work

Clicking the *Find Organizations* and *Find Connected Orgs* buttons will start the computation of organizations or connected organizations respectively. The result of this computation will be presented in the organization tab.

While OrgWeb is doing computational work, the side-bar displays at its bottom left corner a small orange spinning circle and the label *Running* indicating that the application is doing work in the background.

Settings Menu

Clicking on the cog icon in the top right corner of OrgWeb will bring up the *Settings* menu in a pop-up. Currently the only option to configure is to enable or disable *Verbose Mode*. If this is activated, the JavaScript console in the developer tools of one's browser will fill up with intermediate results and responses from the hash object. This however, significantly slows down the computation of organization.

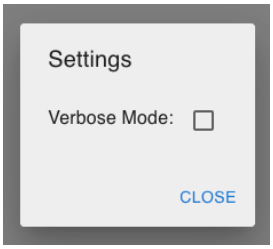


Figure 3.21: Settings Menu

Editor Tab

The *Editor* tab consists of an integration of the *Monaco-Editor* with custom syntax highlighting and a non-visible *Dropzone*.

If the user would 'drag' a file – of the acceptable type – over the *Editor* and 'drop' it there, the text would be updated. The *Editor* also allows the creation of completely new reaction networks both in **SBML** and **REA** changing the syntax highlighting based on the xml-definition line[18].

[18]first line of a xml file looks something like: `<?xml version="1.0" encoding="UTF-8"?>`

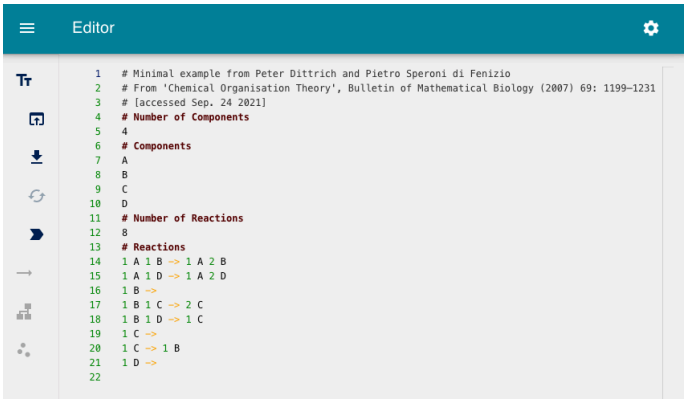


Figure 3.22: The *Editor* tab with the *simpleNetwork.rea*[12] example file loaded. Syntax highlighting for a **REA** file.

Reaction Network Tab

To visually verify that all reactions and species were complete and correctly typed in the *Editor* or imported from a file OrgWeb has the ability to generate a simple reaction network graph from the read network text. This graph is displayed in the *Reaction Network* tab. Its only component is a *cytoscape.js* view. When a reaction network graph was generated and dispatched to the store, *cytoscape.js* will then display it as shown in Figure 3.23. A graph representing the reaction network will

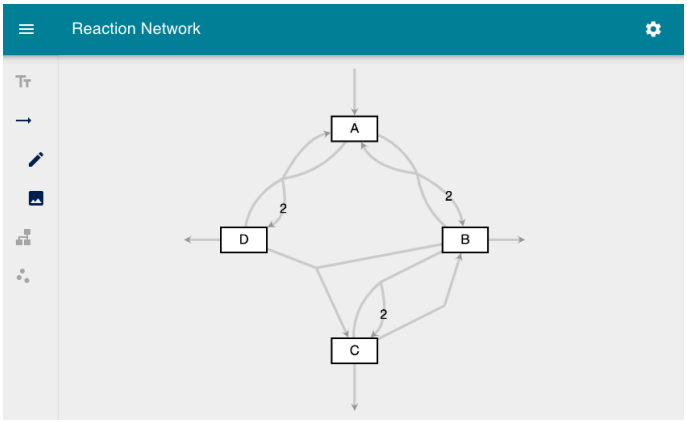


Figure 3.23: The *Reaction Network* tab with the *simpleNetwork*[12] (modified to show all edge types) example displayed as a reaction network graph.

be displayed consisting of two types of nodes and four types of edges. The first and more visible node type is the species node. Each species defined corresponds to one species node. These have a label – the species name – and a black border surrounding it. The second type is the invisible reaction node. Each reaction has one of these. It is located at the intersection between all input and output edges of a reaction. Both of these nodes can be grabbed and moved freely in the *cytoscape.js* view.

The four types of edges are inflow, outflow, reactant and product edges. Inflow and outflow edges represent reactions where one side of the reaction is empty (in- and outflow of the system). These are visualised as arrows directly going in and out of species node and not entering a reaction node. The reactant edges are edges going out (no arrow tip) of a species node and in to a reaction node. They can be labeled with a number, this number is equivalent to the stoichiometric coefficient of the species in the reaction. If they are not labeled, the stoichiometric coefficient is 1.0. Product edges are analog to the reactant edges, however they are drawn with an arrow like tip.

Lattice Tab

Similar to the *Reaction Network* tab consists the *Lattice* tab of one *cytoscape.js* component. This time there are only one type of nodes and one type of edges. A node represents one organization and an edge is drawn between two nodes if the organization corresponding to the node above can be created from the organization below with addition of novelty species. This is best shown through the *Node Label*

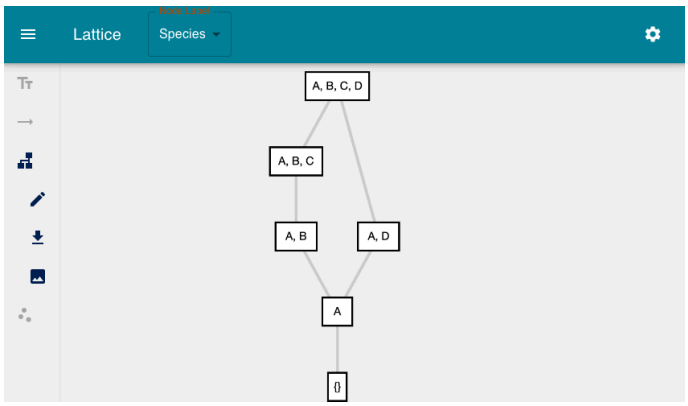


Figure 3.24: The *Reaction Network* tab with the *simpleNetwork*[12] example displayed as a reaction network graph.

of only showing novelty species as mentioned above.
Nodes in this lattice graph can only be moved horizontally. This is because the organizations are fixed in the vertical axis by their cardinality. The Nodes width will also update with switching the *Node Label* so all nodes provide enough space for the displayed labels.

Organization Tab

The *Organization* tab contains a `List` element. The first element in the `List` is a header, informing the user about the number of found elements. The next elements are a dropdown elements for each found organization. The dropdowns shows the index of each organization and if expanded host a list of all species included in the organization set. It also displays a list of all reactions consisting of species of the organization.
If the `List` expands over the windows lower bounds, it will become a scrollable `List`.

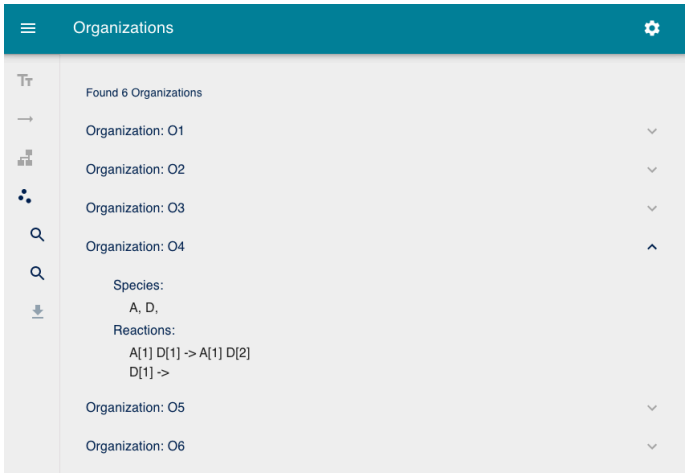


Figure 3.25: The *Organization List* tab with the found organizations of the *simpleNetwork*[12]. Organization *O4* is expanded.

3.2.2 Input/Output

OrgWeb allows for a range of files being input and output. This section will cover what which file is and how it can be used.

SBML

The *systems biology markup language* (SBML) is an open, xml-based format. It was designed to represent biochemical reaction networks by M. Hucka *et al.* in 2003.[22][19]

For the SBML file to work with OrgWeb it needs to have at least some tags specified.
A `<xml>`-header needs to be set. Then the `<sbml>` tag is declared which contains a `<model>` tag. This will include three list items. First a `<listOfCompartments>` containing at least a dummy `<compartment />`. Second a `<listOfSpecies>` is specified containing all the `<species id="" name="" compartment=""/>` elements. Finally the `<listOfReactions>` is defined with all the `<reaction id="" reversable=false>` defined. A `<reaction>` component is comprised of a `<listOfReactants>` with `<speciesReference species="" stoichiometry=1/>` and analog the `<listOfProducts>`.[20]

rea

The rea file format is an in-house format designed by the BioSys-Group at the Friedrich-Schiller-University, Jena. It consists of four parts.

A `# Number of Components` line indicates the first part. The next line states how many species are in the Network. The line below that indicates the next part of the file with the statement `# Components` followed by as many lines as stated in the first part with the name of each species. `# Number of Reactions` is

[19] for an exact specification of SBML see http://www.sbml.org
[20] see **Appendix 5.12** for the *simpleNetwork.xml*

analog to `# Number of Components`. `# Reactions` marks the final section of the file. Each following line contains exactly one reaction. A reaction contains reactants and products. Each reactant and product also has a stoichiometric coefficient in front of it. Separating reactants from products is a `->`. Everything in the reaction line is separated by a space. An Example reaction would be: `1 A 1 B -> 1 A 2 B`. This would be equivalent to the reaction $A + B \rightarrow A + 2B$. Leaving one of the sides (reactants or products) empty will result in an inflow or outflow into the system.[21]

Both SBML and rea files can be dragged and dropped directly into the *Editor* tab, written directly in the *Editor* or opened via the *Open File Dialog*. A written or modified file can also be downloaded via the *Save File* button.

SVG

Scalable vector graphics (*svg*) is a xml-based graphics file format.[37] While grid based image formats like PNG, JPEG or BMP are – after decompression – just stored as values per pixel, *svg* – other vector graphics as well – is saved as a set of two dimensional point definitions and edges or curves between them. A shape is then defined by a closed set of curves, edges and points. *Svg* is resolution-independent because of this algebraic approach to drawing images. This makes them potentially smaller in file size, since an image does not need to contain a definition for each pixel, but only for point pairs and how they are connected.[9]

Svg files are used to export the drawn graphs. Their resolution independence makes them especially suitable for exports of very large lattices or reaction networks.

OrgML/Lattice

OrgML and Lattice files are both home-grown file types developed out of the necessity to store the results of organization computing.
The Lattice or `.ltc` file is a raw text file which contains a List of nodes in the style of `Org.Num= 2 #Elem=2 { A B }` (`A` and `B` reference species id's) and a list of links in the style of `( 0 Below 1 ) Upload= 0 Download= 0.` `( 0 Below 1 )` represents a link between `Org.Num= 0` and `Org.Num= 1`. `Up-` and `Download` are specified for flow directions. However OrgWeb does not take these into account.[22]
OrgML is a SBML/xml-based file type. It contains a xml-header line and then defines a `orgml` component. This contains a `<lattice>` component, which is comprised of a `<listOfOrganizations>` component and a `<listOfLinks>` component. `<listOfLinks>` contains a number of links which are defined as follows: `<link downLink="0.0" organizationAbove="1" organizationBelow="0" upLink="0.0"/>`. The numbers for above and below are again id's of organizations defined in the `<listOfOrganizations>` component. An `<organization id="" name="">` component contains besides an `id` and `name` a set of `<speciesReference>` which ties

[21] see **Appendix 5.13** for the *simpleNetwork.rea*
[22] For an example see **Appendix 5.15**

up to the original sbml/rea file.[23]

Both OrgML and Lattice need a corresponding SBML or rea file to be viewed in OrgViewer. OrgWeb currently does not support viewing of these files only exporting.

3.3 Testing

While implementing the constructive-algorithm used to compute organizations some problems arose. Two main problems are noteworthy, which leave some questions for further discussions.

As stated in the Chapter about Implementation, to test a semi-organization for self-maintenance, a LP-problem is defined and a basis solution is computed with the simplex algorithm. In an earlier implementation of OrgWeb, the commands for adding rows and adding columns to the LP matrix were mixed up. Instead of reactions as basis variables species were defined. Interestingly this still found all organizations for some test cases (mainly the simpleNetwork[12], E. coli and lambda phage[7]), but for others it generated different results. However it always got fewer results than expected.

As the first problem was fixed, a second problem arose. Now for all the test cases OrgWeb would compute the correct results. However, when computing results for Hegele[21], OrgWeb would again compute less organizations than OrgTools. This time OrgWeb would not find enough semi-organizations (this was also true for connected (semi-)organizations). This was quickly deduced since the networks from Hegele were catalytic flow systems[24]. 'In a catalytic flow system, every semi-organization is an organization' (Lemma 28 Dittrich and Speroni di Fenizio)[12]. These results were computed due to not generating all possible *production sets* through combinations. Could there be a way to reduce the number of sets to check and also find all organizations? And if – presuming there is – how can one test the solution for correctness? Especially since OrgWeb was correct in all cases until it was not.

These two mishaps while implementing demonstrate the need for some sort of validation mechanism of results in the future.

[23]For an example see see **Appendix 5.14**

[24]see **Appendix 5.2** for definition.

4. Discussion and Conclusion

4.1 Summary

This paper has re-capitulated what the *Chemical Organization Theory* is and how one would find these organizations given a reaction network. It presented some tools that were used to build a web application and discussed its implementation using TypeScript. Then covering the UI and I/O of the application and pointing to some problems in testing the implementation of the algorithm – used for computing the organizations – for correctness and in general how to debug he web application. Concluding with a short description of a use case of the application.

4.2 Comparison to OrgTools

Comparing OrgWeb to OrgTools the first thing that springs to mind is, OrgWeb is an all in one package, OrgTools is a host of programs. However OrgTools has some functionalities OrgWeb has not, yet. OrgTools' network analysis (not the computation of organizations) is a complete part that was not implemented in OrgWeb. Or the possibility to choose between two algorithms and a heuristic, while OrgWeb forces one to use the *Constructive Algorithm*.

However, both packages run the *Constructive Algorithm*, so Table 4.1 will show the runtime comparison. The times were measured by running each problem five times, taking the best one. The benchmark problems were taken from Centler *et al.*[7]. Problems the algorithm did not terminate while computing in 2008 were not benchmarked.

The system on which was benchmarked had the following specifications: Ubuntu 20.04.02 64-bit, 4GB RAM, i7-4870HQ 1×2.5GHz (VirtualBox VM on mid 2014 MacBook Pro).

As the runtimes show, OrgTools is – as expected – faster on the same machine. However the gap between both runtimes is not too large, as to neglect the usage of OrgWeb. The easy entry to OrgWeb – visiting a website compared to installing dependencies and setting environment variables – is a huge improvement over OrgTools.

	Species/Reactions	OrgWeb	OrgTools
Dry Mars, night	7/15	0.3s	0.3s
Dry Mars, day	7/16	0.2s	0.3s
Mars, day	31/104	14.6s	4.3s
Lambda phage	55/81	0.4s	0.5s
Central E.coli	92/168	2.2s	0.7s
EGFR 1	7/3	0.5s	0.3s
EGFR 2	8/5	0.5s	0.3s
EGFR 3	10/8	0.6	0.4s
EGFR 4	19/21	69.1s	21.0s
EGFR 10	356/3749	-	61.2s

Table 4.1: Comparison of the runtimes between OrgWeb and OrgTools. EGFR 10 did not finish in 30 minutes of runtime on the Linux VM. However it finished on the MacBook Pro itself, but that would not count towards a fair comparison.

4.3 Limitations of OrgWeb

OrgWeb has some limitations. Some are technical limitations, some are due to the implementation.

4.3.1 Multi-Threading

Multi-Threading in JavaScript is possible through the *Web Workers API*. The MDN docs state 'Data is sent between workers and the main thread via a system of messages[...]. The data is copied rather than shared.'[31]. Workers also have their own scope, so they do not have access to the same variables as other workers or the main thread.[31]

Using workers for computing semi-organizations is possible in theory. However it is not practical. A reason why the constructive algorithm is usable (considering computing time), is that the hash prevents the algorithm from checking the same set over and over again. Workers do not share this hash. Every time a worker starts to work, the hash is cleared or is at least not as complete as it could be. Workers will check sets multiple times and thus be slower. This is especially noticeable in big networks with few organizations.

For now OrgWeb is not multi-threaded. It offloads computing into a second thread, but does not utilise multiple threads for that.

4.3.2 Progress Feedback

A big limitation is the minimal progress feedback. There is currently no reliable way of knowing when a computation will terminate or if it will terminate in the near future. A possible solution would be to calculate the connected (semi-)organizations first in order to then estimate an upper bound of possibilities. However there would be no feedback for calculating the connected (semi-)organizations, which although

in most cases fast, can take a long time in some cases, resulting again in the original problem.

4.4 Further Work

OrgWeb is usable and comprises all target functionalities. However there is always something that could be added. This subsection will address some thoughts that could expand the features for network analysis.

4.4.1 OrgML-/Lattice-Viewer

Currently OrgWeb is not able to load in and display a OrgML or lattice file. Not that it would not be capable, but there was limited time to implement a functionality for reading and combining these files with their matching SBML or rea files.

4.4.2 Temporary Files

Currently OrgWeb has no way of providing intermediate results from which a future session could pick up the computation again. This is an ability of OrgTools and should also be considered in future updates to OrgWeb.

To add this functionality one should make a host of interrupt commands available to the user. These would then stop the current computation, possibly after finishing the current cardinality level or immediately and then trigger a download of a temporary results file.

4.4.3 Keyboard Short-Cuts

The user should be able to switch tabs and start functions via a keyboardshort-cut. There are `onKeyDown`-Events defined in JavaScript.[29]

4.4.4 Partial Multi-Threading

As mentioned above, everything is currently done in one thread. For computing the semi-organizations there is currently no other way in the web. However, when analysing a network at genome size, the simplex algorithm takes its time. Since there is no need for solutions from prior calculations, this could be offloaded in multiple workers.

4.4.5 WebAssembly for Computing Semi-Organizations

Comparing the runtime between OrgWeb and OrgTools on the same system is another example of how much faster compiled code runs. The runtime of OrgWeb could be further optimized if it could be compiled to WebAssembly and not be interpreted[42]. As mentioned above WebAssembly can be compiled from many

languages. C/C++ (emscripten), as well as AssemblyScript (a variant of TypeScript) can be compiled to WebAssembly binary format. Since there is a code base for both those programming languages it should not be too difficult.

Since WebAssembly can also be run as a command-line-interface application this could result in a single codebase for web application and CLI.

Note: WebAssembly is still in development. Not all functionality is available in all languages and all browsers[48][43].

4.4.6 Further possibilities

Going on, OrgWeb is just a starting point for more development. Possible extensions could take kinetic laws into consideration. Or they could implement the heuristic algorithm for very fast rough results. They could also implement following trajectories imported from ordinal differential equations (ODEs). *cytoscape.js* allows for animations, so possibly the change described in an ODE could be animated.

Extending the customisation abilities and possible filter functions would make perfect sense.

Many things can be done in the future, but OrgWeb can handle many applications right now. I am eager to see what use cases it will handle, if and how it will be extended.

Bibliography

[1] Abramov,D. and the Redux documentation authors. *Immutable Data — Redux*. [Accessed October, 6th 2021]. 2021. URL: https://redux.js.org/faq/immutable-data#why-is-immutability-required-by-redux.

[2] Abramov,D. and the Redux documentation authors. *React Redux — React Redux*. [Accessed October, 6th 2021]. 2021. URL: https://react-redux.js.org/.

[3] Abramov,D. and the Redux documentation authors. *Redux - A predictable state container for JavaScript apps. — Redux*. [Accessed October, 6th 2021]. 2021. URL: https://redux.js.org.

[4] Abramov,D. and the Redux documentation authors. *Structuring Reducers — Redux*. [Accessed October, 12th 2021]. 2021. URL: https://redux.js.org/usage/structuring-reducers/structuring-reducers.

[5] Arney,T. and react-app-rewired contributors. *timarney/react-app-rewired: Override create-react-app webpack configs without ejecting.* [Accessed October, 7th 2021]. 2021. URL: https://github.com/timarney/react-app-rewired.

[6] Baharev,A. *Re: [Help-glpk] Perform phase 1 simplex only*. [Accessed October, 12th 2021]. 2008. URL: https://lists.gnu.org/archive/html/help-glpk/2008-05/msg00031.html.

[7] Centler,F. Kaleta,C. Speroni di Fenizio,P. Dittrich,P. "Computing chemical organizations in biological networks". In: *Bioinformatics* Vol. 24 no. 14 (2008), pp. 1611–1618.

[8] Centler,F. Speroni di Fenizio,P. Matsumaru,N. Dittrich,P. "Chemical Organizations in the Central Sugar Metabolism of Escherichia coli". In: *Mathematical Modeling of Biological Systems* Vol. 1 (2007), pp. 105–119.

[9] Chapman,N.P. Chapman,J. *Digital multimedia*. New York: John Wiley and Sons, 2000.

[10] Charles University, Prague. *GNU Linear Programming Kit - Reference Manual*. [Accessed September, 15th 2021]. 2010. URL: https://kam.mff.cuni.cz/~elias/glpk.pdf.

[11] Dantzig,G.B. "The Dantzig simplex method for linear programming." In: *IEEExplore* Vol. 2 Issue 1 (1947), pp. 234–241.

[12] Dittrich,P. Speroni di Fenizio,P. "Chemical Organisation Theory". In: *Bulletin of Mathematical Biology* 69 (2007), pp. 1199–1231.

[13] Emscripten contributors. *Main – Emscripten 2.0.31 (dev) documentation.* [Accessed October, 6th 2021]. 2021. URL: `https://www.assemblyscript.org/frequently-asked-questions.html`.

[14] Facebook Inc. *Create React App.* [Accessed October, 7th 2021]. 2021. URL: `https://create-react-app.dev`.

[15] Facebook Inc. *Intoducing JSX - React.* [Accessed October, 8th 2021]. 2021. URL: `https://reactjs.org/docs/introducing-jsx.html`.

[16] Fontana,W. Buss,L.W. ""THE ARRIVAL OF THE FITTEST": TOWARD A THEORY OF BIOLOGICAL ORGANIZATION". In: *Bulletin of Mathematical Biolog* Vol. 56 Issue 1 (1994), pp. 1–64.

[17] Franz,M. Cheung,M. Adel,H. Li,L. YikaJ Kortchmar,S. Fjukstad,B. Ephraim,J. N8th8n8el Blumenfeld,Z. *cytoscape/cytoscape.js-dagre: The Dagre layout for DAGs and trees for Cytoscape.js.* [Accessed October, 12th 2021]. 2020. URL: `https://github.com/cytoscape/cytoscape.js-dagre/tree/v2.3.2`.

[18] Franz,M. Cheung,M. Li,A. Qingqing. *cytoscape/cytoscape.js-automove: An extension for Cytoscape.js that automatically updates node positions based on specified rules.* [Accessed October, 12th 2021]. 2019. URL: `https://github.com/cytoscape/cytoscape.js-automove`.

[19] Franz,M. Lopes,C.T. Huck,G. Dong,Y. Sumer,O. Bader,G.D. "Cytoscape.js: a graph theory library for visualisation and analysis". In: *Bioinformatics* Vol. 32 Issue 2 (2015), pp. 309–3011.

[20] Grey,E. and FileSaver.js contributors. *eligrey/FileSaver.js: An HTML5 saveAs() FileSaver implementation.* [Accessed October, 7th 2021]. 2020. URL: `https://github.com/eligrey/FileSaver.js#readme`.

[21] S. Hegele. "Identifikation autopoietischer Strukturen in sozialer Kommunikation mit der chemischen Organisationstheorie". 2021.

[22] Hucka,M. Finney,A. Sauro,H.M. Bolouri,H. Doyle,J.C. Kitano,H. Arkin,A.P. Bornstein,B.J. Bray,D. Cornish-Bowden,A. Cuellar,A.A. Dronov,S. Gilles,E.D. Ginkel,M. Gor,V. Goryanin,I.I. Hedley,W.J. Hodgman,T.C. Hofmeyr,J.-H. Hunter,P.J. Juty,N. S. Kasberger,J. L. Kremling,A. Kummer,U. LeNovre,N. Loew,L.M. Lucio,D. Mendes,P. Minch,E. Mjolsness,E.D. Nakayama,Y. Nelson,M.R. Nielsen,P.F. Sakurada,T. Schaff,J.C. Shapiro,B.E. Shimizu,T.S. Spence,H.D. Stelling,J. Takahashi,K. Tomita,M. Wagner,J. Wang,J. and the rest of the SBML Forum. "The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models". In: *Bioinformatics* Vol. 19 Issue 4 (2003), pp. 524–531.

[23] kinimesi. *kinimesi/cytoscape-svg:A Cytoscape.js extension to export the current graph view as an SVG.* [Accessed October, 12th 2021]. 2020. URL: `https://github.com/kinimesi/cytoscape-svg`.

[24] Material-UI SAS. *Material icons - MUI.* [Accessed October, 8th 2021]. 2021. URL: `https://mui.com/components/material-icons/`.

[25] Material-UI SAS. *The React component library you always wanted.* [Accessed October, 6th 2021]. 2021. URL: `https://mui.com`.

[26] Microsoft Corporation. *Monaco Editor*. [Accessed October, 7th 2021]. 2021. URL: https://microsoft.github.io/monaco-editor/index.html.

[27] Microsoft Corporation. *Monaco Editor Playground*. [Accessed October, 7th 2021]. 2021. URL: https://microsoft.github.io/monaco-editor/playground.html#extending-language-services-custom-languages.

[28] Mozilla and individual contributors. *Equality (==) - JavaScript — MDN*. [Accessed October, 8th 2021]. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Equality.

[29] Mozilla and individual contributors. *GlobalEventHandlers.onkeydown - Web APIs — MDN*. [Accessed October, 7th 2021]. 2021.

[30] Mozilla and individual contributors. *Strict equality (===) - JavaScript — MDN*. [Accessed October, 8th 2021]. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Strict_equality.

[31] Mozilla and individual contributors. *Web Workers API - Web APIs — MDN*. [Accessed October, 7th 2021]. 2021.

[32] Murray,J.D. *Mathematical Biology I, An Introduction*. Heidelberg: Springer-Verlag, 2004.

[33] Node.js contributors, Open JS Foundation, Joynet Inc. *Node.js*. [Accessed October, 5th 2021]. 2021. URL: https://nodejs.org/en/.

[34] Norris,A. Barry,J. Popov,V. Ivanov,V. *alisternorris/file-dialog: Trigger the upload file dialog directly from your code easily*. [Accessed October, 7th 2021]. 2019. URL: https://github.com/alisternorris/file-dialog#readme.

[35] npm contributors, npm Inc. *About npm — npm Docs*. [Accessed October, 5th 2021]. 2021. URL: https://docs.npmjs.com/about-npm.

[36] Pirman,K. *React Hooks - CodeSandbox*. Accessed October, 12th 2021. 2021. URL: https://codesandbox.io/s/b0ntj.

[37] Quint,A. "Scalable vector graphics". In: *IEEE MultiMedia* 10.3 (2003), pp. 99–102. DOI: 10.1109/MMUL.2003.1218261.

[38] react-dropzone contributors. *react-dropzone*. [Accessed October, 7th 2021]. 2021. URL: https://react-dropzone.js.org.

[39] React.js contributors, Facebook Inc. *Tutorial: Intro to React - React*. [Accessed October, 5th 2021]. 2021. URL: https://reactjs.org/tutorial/tutorial.html#what-is-react.

[40] Surma. *React + Redux + Comlink = Off-main-thread - surma.dev*. [Accessed October, 12th 2021]. 2019. URL: https://surma.dev/things/react-redux-comlink/.

[41] Surma, Google Chrome Labs. *GoogleChromeLabs/comlink: Comlink makes WebWorkers enjoyable.* [Accessed October, 7th 2021]. 2021. URL: https://github.com/GoogleChromeLabs/comlink.

[42] The AssemblyScript Project. *Frequently asked questions — The AssemblyScript Book*. [Accessed October, 6th 2021]. 2021. URL: https://www.assemblyscript.org/frequently-asked-questions.html.

[43] The AssemblyScript Project. *Implementation status — The AssemblyScript Book*. [Accessed October, 7th 2021]. 2021. URL: `https://www.assemblyscript.org/status.html`.

[44] TypeScript contributors, Microsoft Corporation. *TypeScript: JavaScript With Syntax For Types*. [Accessed October, 5th 2021]. 2021. URL: `https://www.typescriptlang.org`.

[45] V8 contributors, Google LLC. *V8 JavaScript engine*. [Accessed October, 6th 2021]. 2021. URL: `https://v8.dev/`.

[46] WebAssembly contributors. *FAQ - WebAssembly*. [Accessed October, 6th 2021]. 2021. URL: `https://webassembly.org/docs/faq/`.

[47] WebAssembly contributors. *I want to... - WebAssembly*. [Accessed October, 6th 2021]. 2021. URL: `https://webassembly.org/getting-started/developers-guide/`.

[48] WebAssembly contributors. *Roadmap - WebAssembly*. [Accessed October, 7th 2021]. 2021. URL: `https://webassembly.org/roadmap/`.

[49] WebAssembly contributors. *WebAssembly*. [Accessed October, 6th 2021]. 2021. URL: `https://webassembly.org/`.

[50] WebPack contributors. *webpack-contrib/worker-loader: webpack-contrib/worker-loader*. [Accessed October, 7th 2021]. 2021. URL: `https://github.com/webpack-contrib/worker-loader`.

[51] WebPack contributors, Open JS Foundation. *webpack*. [Accessed October, 7th 2021]. 2021. URL: `https://webpack.js.org`.

[52] Wikipedia. *Lattice (order) - Wikipedia*. [Accessed October, 15th 2021]. 2021. URL: `https://en.wikipedia.org/wiki/Lattice_(order)`.

[53] Yousuf and xml-js contributors. *nashwaan/xml-js: Converter utility between XML text and Javascript object / JSON text.* [Accessed October, 7th 2021]. 2019. URL: `https://github.com/nashwaan/xml-js#readme`.

5. Appendix

5.1 Difference == and ===

== is called the *equality* operator. Since JavaScript has no types it tries to convert types and the check if the values are the same.
=== is called the *strict equality* operator. This will not convert types to check for equality.[28][30]

Figure 5.4 shows the two operators in code.

5.2 Catalytic Flow Systems

'An algebraic chemistry $\langle \mathcal{M}\mathcal{R} \rangle$ is called a catalytic flow system, if for all molecules $i \in \mathcal{M}$: (1) there exists a reaction $(\{i\} \to \emptyset) \in \mathcal{R}$, and (2) there does not exist a reaction $(A \to B) \in \mathcal{R}$ with $(A \to B) \neq (\{i\} \to \emptyset)$ and $\#(i \in A) > \#(i \in B)$.'(Dittrich and Speroni di Fenizio, 2007, p. 1211 in [12])

Source Code

One can find the source code at the gitlab repository:
https://git.rz.uni-jena.de/go53doc/chemicalorganisation-webtool/-/tree/main/

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web-App to calculate chemical organizations."
    />
    <!--
     [...]
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      [...]
    -->
    <title>Chemical Organization Theorie</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      [...]
    -->
  </body>
</html>
```

Figure 5.1: The `./public/index.html` file.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web-App to calculate chemical organizations."
    />
    <!--
      [...]
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--
      [...]
    -->
    <title>Chemical Organization Theorie</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      [...]
    -->
  </body>
</html>
```

Figure 5.2: The `./public/index.html` file.

```tsx
import React from 'react';
import ReactDOM from 'react-dom';
import { store } from './app/store';
import { Provider } from 'react-redux';
import Main from './Main';

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <Main />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);
```

Figure 5.3: The `./src/index.tsx` file.

```tsx
"1" == 1 // true
2 == 3-1 // true

"1" === 1 // false
2 === 3-1 // true
```

Figure 5.4: Showing the difference between equality operators.

```
import { FormControl, InputLabel, MenuItem, Select, SelectChangeEvent } from "@mui/
    material";
import { EdgeDefinition } from "cytoscape";
import React from "react";
import { useAppDispatch, useAppSelector } from "../app/hooks";
import { NodeValues, setNodeView } from "../app/state/appState";
import generateLattice from "../features/lattice/generateGraph";

export function NodeViewDropdown() {
    const dispatch = useAppDispatch();
    const label = useAppSelector(state => state.app_reducer.nodeView);
    const lattice = useAppSelector(state => state.app_reducer.latticeGraph);
    const numOfOrgs = useAppSelector(state => state.app_reducer.orgList.length);

    const handleChange = (event: SelectChangeEvent) => {
        dispatch(setNodeView(parseInt(event.target.value)));
        const nodes = lattice.slice(0, numOfOrgs);
        const edges = lattice.slice(numOfOrgs) as EdgeDefinition[];
        generateLattice(nodes, edges);
    };
    return (
        <FormControl sx={{ m: 1, minWidth: 80 }}>
            <InputLabel id="demo-simple-select-label" sx={{color: '#BC4A09'}}>Node
                Label</InputLabel>
            <Select
                labelId="demo-simple-select-label"
                id="demo-simple-select"
                value={label.toString()}
                label="Node Label"
                onChange={handleChange}
                sx={{color: '#fff'}}
            >
                <MenuItem value={NodeValues.ALLSPECIES}>Species</MenuItem>
                <MenuItem value={NodeValues.NOVELTY}>Novelty species</MenuItem>
                <MenuItem value={NodeValues.ORGNAMES}>Org index</MenuItem>
            </Select>
        </FormControl>
    );
}
```

Figure 5.5: `./src/components/nodeViewDropdown.tsx` code.

```
const drawerWidth: number = 280;

const Drawer = styled(MuiDrawer, { shouldForwardProp: (prop) =>
    prop !== 'open' })(
     ({ theme, open }) => ({
         '& .MuiDrawer-paper': {
                 position: 'relative',
                 whiteSpace: 'nowrap',
                 width: drawerWidth,
                 transition: theme.transitions.create('width', {
                 easing: theme.transitions.easing.sharp,
                 duration: theme.transitions.duration.enteringScreen
                     ,
             }),
             boxSizing: 'border-box',
             ...(!open && {
                 overflowX: 'hidden',
                 transition: theme.transitions.create('width', {
                     easing: theme.transitions.easing.sharp,
                     duration: theme.transitions.duration.
                         leavingScreen,
                 }),
                 width: theme.spacing(7),
                 [theme.breakpoints.up('sm')]: {
                     width: theme.spacing(9),
                 },
             }),
         },
     }),
);
```

Figure 5.6: The implementation of the styled `Drawer`. Styling the drawer will provide a custom theme, this can be used to set up animations.

```
interface TabPanelProps {
    children?: React.ReactNode;
    index: string;
    value: string;
}

function TabPanel(props: TabPanelProps) {
    const { children, value, index, ...other } = props;

    return (
      <div
        role="tabpanel"
        hidden={value !== index}
        id={`simple-tabpanel-${index}`}
        aria-labelledby={`simple-tab-${index}`}
        style={{height: '100%'}}
        {...other}
      >
        {value === index && (
          children
        )}
      </div>
    );
}
```

Figure 5.7: The implementation of the `TabPanel`.

```
monaco.languages.register({ id: 'rea' });

monaco.languages.setMonarchTokensProvider('rea', {
    tokenizer: {
        root: [
            [/->/, "reaction"],
            [/# Components/, "components"],
            [/# Number of Components/, "component-count"],
            [/# Reactions/, "reactions"],
            [/# Number of Reactions/, "reaction-count"],
            [/[0-9.]+ /, "stoichiometry"],
            [/#.*/, "comment"],
        ]
    }
});

monaco.editor.defineTheme('org-theme-rea', {
    base: 'vs',
    inherit: true,
    rules: [
        { token: 'comment', foreground: '303030' },
        { token: 'components', foreground: '550000', fontStyle: 'bold' },
        { token: 'component-count', foreground: '550000', fontStyle: 'bold' },
        { token: 'reactions', foreground: '550000', fontStyle: 'bold' },
        { token: 'reaction-count', foreground: '550000', fontStyle: 'bold' },
        { token: 'reaction', foreground: 'FFA500' },
        { token: 'stoichiometry', foreground: '008800' },
    ],
    colors: {
        'editor.background': '#eee',
        'editor.lineHighlightBackground': '#eee',
        'editorLineNumber.foreground': '#008800',
    }
});
monaco.editor.defineTheme('org-theme-xml', {
    base: 'vs',
    inherit: true,
    rules: [],
    colors: {
        'editor.background': '#eee',
        'editor.lineHighlightBackground': '#eee',
        'editorLineNumber.foreground': '#008800',
    }
});
```

Figure 5.8: Monaco Editor Theme definitions.

```
import { Species } from "./species";

export default class MultiSet {
    hashString: string = "";
    cardinality: number;
    species: Array<Species>;
    booleanArray: Array<boolean>;
    constructor(species: Array<Species>, numberOfSpecies: number) {
        this.species = species;
        this.cardinality = species.length;
        this.booleanArray = new Array<boolean>(numberOfSpecies).fill(false);
        for(let i = 0, k = species.length; i < k; i++) {
            const id: number = species[i].index;
            this.booleanArray[id] = true;
        }
        this.hashString = this.arrayToString();
    }
    hasSpecies(species: Species): boolean {
        return this.booleanArray[species.index];
    }
    addSpecies(species: Species): boolean {
        if(this.hasSpecies(species) === false) {
            this.species.push(species);
            this.cardinality = this.species.length;
            this.booleanArray[species.index] = true;
            this.hashString = this.arrayToString();
            return true;
        }
        return false;
    }
    getSpecies(index: number): Species {
        return this.species[index];
    }
    getSpeciesFromID(id: number): Species {
        for(let i = 0; i < this.cardinality; i++) {
            if(this.species[i].index === id) {
                return this.species[i];
            }
        }
        return this.species[0];
    }
    getHashString(): string {
        return this.hashString;
    }
    getBooleanArray(): Array<boolean> {
        return this.booleanArray;
    }
    getCardinality(): number {
        return this.cardinality;
    }
    toString(): string {
        let str: string = "{";
        for(let i = 0, k = this.cardinality; i < k; i++) {
            str = str + " [" + this.species[i].toString() + "]";
        }
        str = str + " }"
        return str;
    }
    arrayToString(): string {
        let str = "";
        for (let i = 0, k = this.booleanArray.length; i < k; i++) {
            if(this.booleanArray[i] === true) {
                str = str + "1";
            } else {
                str = str + "0";
            }
        }
        return str;
    }
}
```

Figure 5.9: The `./src/app/classes/multiSet.ts` source code.

```
import MultiSet from "./multiSet";

export default class Hash {

    hash: Map<string, number>[];

    verbose: boolean;

    constructor(numberOfSpecies: number, verbose: boolean, hash?: Map<string,
        number>[]) {
        if (hash) {
            this.hash = hash;
        } else {
            this.hash = [];
        }
        this.verbose = verbose;
    }

    freeHash(): void {
        this.hash = [];
    }

    insert(set: MultiSet): boolean {
        const size: number = set.getCardinality();
        const hashSrting = set.getHashString();
        if(this.hash[size] && this.hash[size].has(hashSrting)) {

            if(this.verbose) {console.debug('Set already processed.', [size,
                hashSrting]);}

            return false;
        } else {
            if(!this.hash[size]){
                this.hash[size] = new Map<string, number>();
            }
            this.hash[size].set(hashSrting, 1);
            if(this.verbose) {console.debug('Set inserted.', [size, hashSrting]);}
            return true;
        }
    }

    freeHashBelow(size: number): void {
        for(let i = 0; i < size; i++) {
            if(this.hash[i]) {
                this.hash[i] = new Map<string, number>();
            }
        }

        if(this.verbose) {console.debug('Freed Hash below.', size);}
    }
}
```

Figure 5.10: The `./src/app/classes/hash.ts` source code.

```
export type OrganizationWorker = {
    connected: boolean;
    organizations: Organization[];
    calculateOrganizations(
        connected: boolean,
        verbose: boolean,
        species: Species[],
        reactions: Reaction[],
        callback:(semiOrganizations: number[], verbose: boolean, error: any |
            undefined) => void
    ): void;
    computeSemiOrganizations(
        reactionList: Array<Reaction>,
        speciesList: Array<Species>,
        connected: boolean,
        verbose: boolean
    ): Array<MultiSet>;
    SOsDirectlyAbove(
        current:MultiSet,
        reactionList: Array<Reaction>,
        speciesList: Array<Species>,
        hash: Hash,
        connected: boolean
    ): Array<MultiSet>;
    SOsDirectlyAboveContaining(
        semiOrg: MultiSet,
        speciesSet: MultiSet,
        speciesList: Array<Species>,
        reactionList: Array<Reaction>,
        hash: Hash
    ): Array<MultiSet>;
    getProducedConsumedSpecies(
        consumedSpecies: MultiSet,
        producedSpecies: MultiSet,
        current: MultiSet,
        speciesList:Array<Species>,
        reactionList: Array<Reaction>
    ):void;
    producerSets(set: MultiSet,
        speciesList: Array<Species>,
        reactionList: Array<Reaction>
    ): Array<MultiSet>;
}
```

Figure 5.11: Implementation of the `OrganizationWorker` interface.

```
<?xml version="1.0" encoding="UTF-8"?>
<sbml xmlns="http://www.sbml.org/sbml/level2" xmlns:html="http://www.w3.org/1999/
    xhtml" xmlns:jigcell="http://www.sbml.org/2001/ns/jigcell" xmlns:math="http://
    www.w3.org/1998/Math/MathML" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax
    -ns#" xmlns:sbml="http://www.sbml.org/sbml/level2" xmlns:xlink="http://www.w3.
    org/1999/xlink" level="2" version="1">
  <model>
    <listOfCompartments>
      <compartment id="cell" name="cell" />
    </listOfCompartments>
    <listOfSpecies>
      <species id="a" name="a" compartment="cell" />
      <species id="b" name="b" compartment="cell" />
      <species id="c" name="c" compartment="cell" />
      <species id="d" name="d" compartment="cell" />
    </listOfSpecies>
    <listOfReactions>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="a" />
          <speciesReference species="b" />
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="a" />
          <speciesReference species="b" stoichiometry="2.0" />
        </listOfProducts>
      </reaction>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="a" />
          <speciesReference species="d" />
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="a" />
          <speciesReference species="d" stoichiometry="2.0" />
        </listOfProducts>
      </reaction>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="b" />
          <speciesReference species="c" />
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="c" stoichiometry="2.0" />
        </listOfProducts>
      </reaction>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="c" />
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="b" />
        </listOfProducts>
      </reaction>
```

```
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="b" />
          <speciesReference species="d" />
        </listOfReactants>
        <listOfProducts>
          <speciesReference species="c" />
        </listOfProducts>
      </reaction>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="b" />
        </listOfReactants>
      </reaction>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="c" />
        </listOfReactants>
      </reaction>
      <reaction reversible="false">
        <listOfReactants>
          <speciesReference species="d" />
        </listOfReactants>
      </reaction>
    </listOfReactions>
  </model>
</sbml>
```

Figure 5.12: The *simpleNetwork.xml*[12] example file.

```
# Number of Components
4
# Components
A
B
C
D
# Number of Reactions
8
# Reactions
1 A 1 B -> 1 A 2 B
1 A 1 D -> 1 A 2 D
1 B ->
1 B 1 C -> 2 C
1 B 1 D -> 1 C
1 C ->
1 C -> 1 B
1 D ->
```

Figure 5.13: The *simpleNetwork.rea*[12] example file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<orgml version="0.1" xmlns="http://www.informatik.uni-jena.de/csb/orgml/version0.1"
    >
    <lattice name="">
        <listOfOrganizations>
            <organization id="0" name="O1">
                <set>
                </set>
            </organization>
            <organization id="1" name="O2">
                <set>
                    <speciesReference id="0" species="A"/>
                </set>
            </organization>
            <organization id="2" name="O3">
                <set>
                    <speciesReference id="0" species="A"/>
                    <speciesReference id="1" species="B"/>
                </set>
            </organization>
            <organization id="3" name="O4">
                <set>
                    <speciesReference id="0" species="A"/>
                    <speciesReference id="3" species="D"/>
                </set>
            </organization>
            <organization id="4" name="O5">
                <set>
                    <speciesReference id="0" species="A"/>
                    <speciesReference id="1" species="B"/>
                    <speciesReference id="2" species="C"/>
                </set>
            </organization>
            <organization id="5" name="O6">
                <set>
                    <speciesReference id="0" species="A"/>
                    <speciesReference id="1" species="B"/>
                    <speciesReference id="2" species="C"/>
                    <speciesReference id="3" species="D"/>
                </set>
            </organization>
        </listOfOrganizations>
        <listOfLinks>
            <link downLink="0.000000" organizationAbove="1" organizationBelow="0"
                upLink="0.000000"/>
            <link downLink="0.000000" organizationAbove="2" organizationBelow="1"
                upLink="0.000000"/>
            <link downLink="0.000000" organizationAbove="3" organizationBelow="1"
                upLink="0.000000"/>
            <link downLink="0.000000" organizationAbove="4" organizationBelow="2"
                upLink="0.000000"/>
            <link downLink="0.000000" organizationAbove="5" organizationBelow="3"
                upLink="0.000000"/>
            <link downLink="0.000000" organizationAbove="5" organizationBelow="4"
                upLink="0.000000"/>
        </listOfLinks>
    </lattice>
</orgml>
```

Figure 5.14: The *simpleNetwork*[12] as OrgML file.

```
#N= 6
Org.Num= 0 #Elem=0 { }
Org.Num= 1 #Elem=1 { A }
Org.Num= 2 #Elem=2 { A B }
Org.Num= 3 #Elem=2 { A D }
Org.Num= 4 #Elem=3 { A B C }
Org.Num= 5 #Elem=4 { A B C D }

#L= 6
( 0 Below 1 ) Upload= 0  Download= 0
( 1 Below 2 ) Upload= 0  Download= 0
( 1 Below 3 ) Upload= 0  Download= 0
( 2 Below 4 ) Upload= 0  Download= 0
( 3 Below 5 ) Upload= 0  Download= 0
( 4 Below 5 ) Upload= 0  Download= 0
```

Figure 5.15: The *simpleNetwork*[12] as lattice file.

6. Selbständigkeitserklärung

Ich erkläre, dass ich, Fionn Daire Keogh, die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Seitens des Verfassers bestehen keine Einwände die vorliegende Bachelorarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, den 16.10.2021