LogicVisualizer

Entwicklungsdokumentation

1. Zielbestimmung

Das Ziel dieses Projekts ist es, eine HTML5-basierte Webanwendung (Visualizer) auf Basis von Webcomponents zu entwickeln, die ein anderes System (Simulator), das eine kombinatorische oder sequentielle Schaltung realisiert, visualisieren und steuern soll.
Dazu werden die Werte der Eingangs- und Ausgangsvariablen (binär) im Zeitverlauf dargestellt und bei sequentiellen Schaltungen werden zusätzlich die Zustandsvariablen zusammengefasst als dezimale Zustandsnummer dargestellt.
Die Steuerung erfolgt über das setzen der Eingangsvariablen und das Steuern des Takts:

- Der Nutzer ändert eine Eingangsvariable.
- Diese wird an den Simulator gesendet.
- Der Simulator sendet berechnete Daten zurück an den Visualizer, welcher diese darstellt.

- Es erfolgt eine Taktanweisung, entweder durch den Nutzer direkt oder nach Timerablauf (bei mehreren Takten).
- Der Visualizer schickt eine Taktaufforderung an den Simulator.
- Dieser führt den Takt aus und schickt die Ausgangsbelegung und die Zustandsnummer zurück an den Visualizer, der dann alles darstellt.

1.1. Musskriterien

- Der Nutzer kann vor Simulationsstart eine Initiale Belegung für Zustände setzen, falls der Simulator das unterstützt.
- Der Nutzer kann den Wert der Eingangsvariablen beliebig verändern.
- Der Nutzer kann die Simulation steuern:
    - einmal takten
    - X mal takten (dabei jeder Takt verzögert, Verzögerungszeit einstellbar)
    - zurücksetzen
- Der Nutzer kann im Diagramm horizontal zoomen, indem er STRG gedrückt hält und das Mausrad bewegt.
- Der Nutzer kann einen Simulationsverlauf speichern und später wieder laden. Die Simulation kann dann allerdings nicht fortgesetzt werden
- Der Nutzer kann die Sprache der Benutzeroberfläche umschalten (mindestens Deutsch und Englisch, erweiterbar um beliebige Sprachen), dabei EN defaultmäßig. Dafür wird i18n benutzt.
- Der Nutzer kann durch Doppelklicken im oberen Diagrammbereich eine vertikale Linie einfügen.
- Das System kann sowohl das Verhalten von kombinatorischen als auch sequentiellen Schaltungen visualisieren.
- Das System stellt die Werte der Eingangs- und Ausgangsvariablen sowie der Zustände (dezimal) im Zeitverlauf als Impulsdiagramm dar.
- Das System kann verbotene Belegungen (h*(x), h*(z), h*(x,z)) im Diagramm kennzeichnen, indem in entsprechenden Takten die problematischen Bereiche (x, z) farblich hinterlegt werden.

1.2. Wunschkriterien

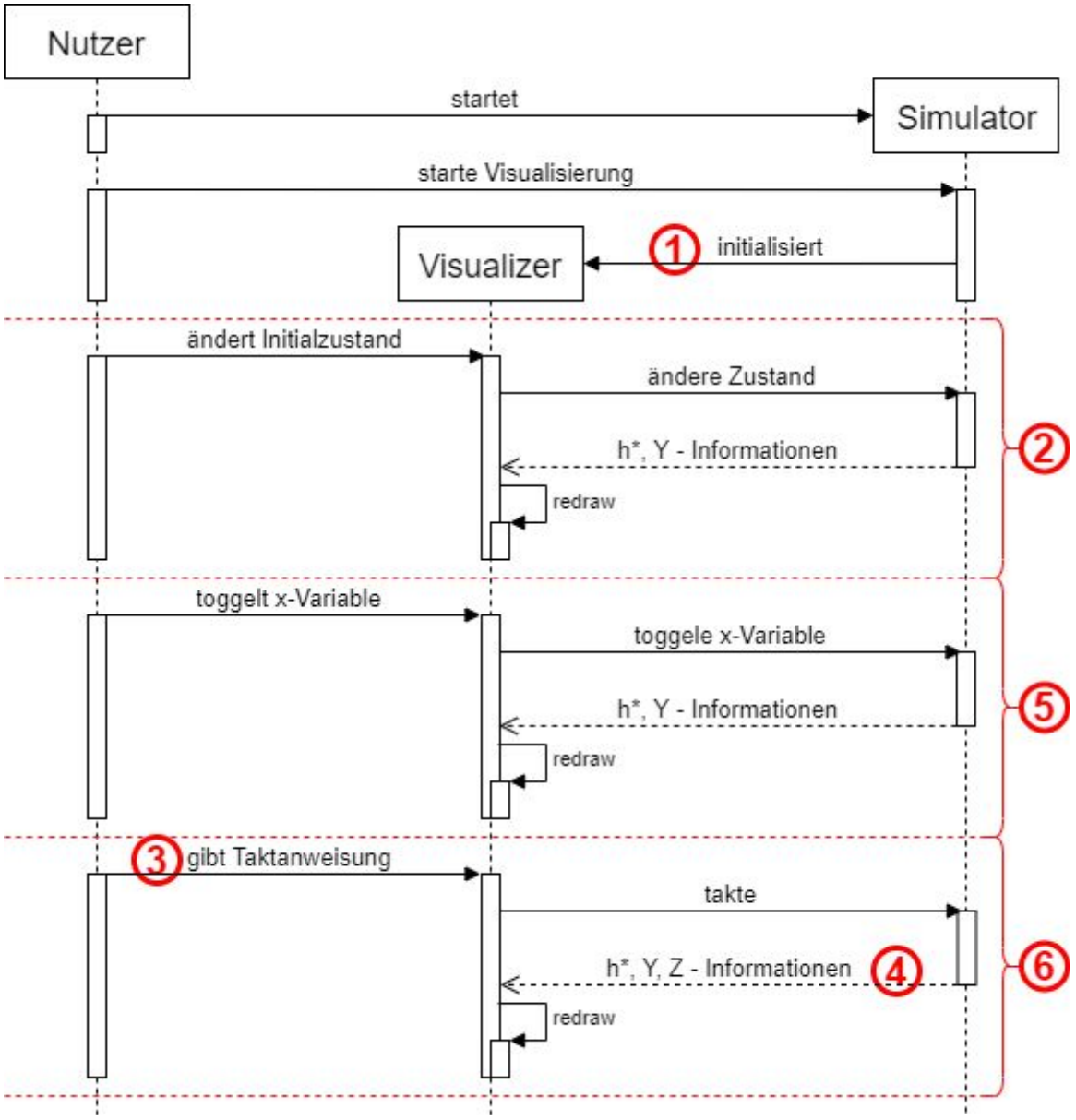
- Der Nutzer kann das aktuell dargestellte Diagramm in voller Breite als Bild (SVG, PNG) exportieren.
- Beim Zoomen ist die x-Koordinate der Maus ein Fixpunkt.
- Der Visualizer liefert einen Simulator mit, sodass man ihn auch standalone benutzen kann.

2. Produktumgebung

Der Visualizer soll in die GOLDi-Website integriert werden. Dort soll er vielseitig nutzbar sein, aber vor allem zur Entwicklung paralleler Automaten dienen.

3. Produkteinsatz

3.1. Sequenzdiagramm



3.2. Erläuterungen

Bei der Initialisierung des Visualizers **(1)** werden folgende Parameter übergeben:

- Ob es sich um eine sequentielle oder kombinatorische Schaltung handelt
- Anzahl und Namen der Teilautomaten, Eingangs- und Ausgangsvariablen
- Initialwerte der Eingangs- und Ausgangsvariablen sowie Initialzustände der Teilautomaten
- Ob der Simulator das Ändern der Teilautomatenzustände vor Simulationsstart (bzw. nach einem Reset) unterstützt

Der Teilablauf **(2)** wird nur ausgeführt, falls vorher bei der Initialisierung übergeben wurde, dass der Simulator dies unterstützt. Falls das so ist, kann er beliebig oft ausgeführt werden, bis zum ersten mal eine Taktanweisung ausgelöst wird.

(3) kann entweder durch Klicken des Takt/Berechnung-Buttons oder durch Timerablauf (bei Mehr-Takt-Ausführung) ausgelöst werden. Handelt es sich um eine kombinatorische Schaltung, ist die Taktanweisung als Berechnungsanweisung zu verstehen. In diesem Fall wird bei **(4)** natürlich keine Z-Information zurückgeschickt, da keine existiert.

Durch Klicken des Reset-Buttons wird der Ablauf auf den Beginn von **(2)** zurückgesetzt.

Der Teilablauf **(5)** kann nicht während Mehr-Takt-Ausführung stattfinden.

Die Teilabläufe **(5)** und **(6)** können beliebig oft und in beliebiger Reihenfolge ausgeführt werden.

3.3. Beispiel

Der LogicVisualizer wird in eine HTML-Seite eingefügt:

```
<script type="module" src="dist/main.js"></script>
<!-- ... -->
<logic-visualizer></logic-visualizer>
```

Die Dispatcher-Funktionen können von “Events.ts” importiert werden.

Der Simulator sendet ein Event zur Initialisierung des Visualizers:

```
dispatchInitializeVisualizerEvent( args );
```

Wobei der Event Name `'lv-initialize'` ist und

```
args = {
            inputs: [{
                  name: "x0",
                  value: "0"
            }, { ... }, ... ],
            states: [{
                  name: "a0",
                  value: "12"
            }, { ... }, ... ],
            outputs: [{
                  name: "y0",
                  value: "1"
```

```
            }, { ... }, ... ],
            hStar: "0",
            canSetState: true
        }
```

Der Nutzer ändert dann den Zustand von a0 auf 1. Dabei wird wieder ein Event gesendet:

```
dispatchUserChangeEvent( args );
```

wobei der Event Name `'lv-user-change'` ist und

```
args = {
            states: [{
                name: "a0",
                value: "1"
            }]
        }
```

Zunächst erhält der Visualizer selbst das Event und updated seine Datenstruktur. Danach erhält der Simulator das Event, ändert intern den Zustand und berechnet evtl. veränderte Ausgaben und h*-Belegung. Danach sendet der Simulator ein Antwort-Event:

```
dispatchSimulatorChangeEvent( args );
```

Wobei der Event Name `'lv-simulator-change'` ist und

```
args = {
            outputs: [{
                name: "y0",
                value: "0"
            }, { ... }, ... ],
            hStar: "1",
            answeringClock: false
        }
```

Alternativ kann dieses Event (oder auch die anderen) manuell erstellt werden (um Zugriff auf den Visualizer-Code vom Simulator zu vermeiden):

```
let args = {
            outputs: [{
                name: "y0",
                value: "0"
            }, { ... }, ... ],
            hStar: "1",
            answeringClock: false
            };
let event = new CustomEvent('lv-simulator-change');
event["args"] = args;
document.dispatchEvent(event);
```

Der Visualizer erhält das Event und updated seine Datenstruktur, woraufhin sich die Darstellung updatet.

Der Nutzer ändert dann x0. Im sequentiellen Fall wird dies sofort an den Simulator gesendet, im kombinatorischen erst bei betätigen des Berechne-Buttons. Es wird wieder per Event kommuniziert:

```
dispatchUserChangeEvent( args );
```

wobei der Event Name `'lv-user-change'` ist und

```
args = {
            inputs: [{
                 name: "x0",
                 value: "1"
            }]
       }
```

Zunächst erhält der Visualizer wieder selbst sein Event und updatet seine Datenstruktur. Danach erhält der Simulator das Event, berechnet evtl. veränderte Ausgaben und h*-Belegung und sendet ein Antwort-Event:

```
dispatchSimulatorChangeEvent( args );
```

Wobei der Event Name `'lv-simulator-change'` ist und

```
args = {
            outputs: [{
                 name: "y0",
                 value: "0"
            }, { ... }, ... ],
            hStar: "0",
            answeringClock: false
       }
```

Der Visualizer erhält das Event und verarbeitet es wie oben.

Dann gibt der Nutzer eine Taktanweisung, indem er auf ‘Takt’ klickt:

```
dispatchClockEvent();
```

Hierauf reagiert zunächst wieder der Visualizer selbst: Es werden in den Datenstrukturen Einträge für den neuen Zeitschritt angelegt. Auch der Simulator erhält das Event, berechnet den neuen Zustand, die neuen Ausgaben und die neue h*-Information und sendet diese im Antwort-Event zurück:

```
dispatchSimulatorChangeEvent( args );
```

Wobei der Event Name `'lv-simulator-change'` ist und

```
args = {
            states: [{
                 name: "a0",
                 value: "1"
            }, { ... }, ... ],
            outputs: [{
                 name: "y0",
                 value: "0"
            }, { ... }, ... ],
            hStar: "0",
```

```
                answeringClock: true
            }
```

Der Visualizer erhält das Event und verarbeitet es wie oben.

4. Produktdaten

Alle wichtigen Informationen werden in einem zentralen Speicherobjekt, dem store (vgl. store.ts) bzw. dessen state, gespeichert. Dies wird durch das Framework Redux ermöglicht.
Hierbei speichert der store einen state und LitElemente, die Informationen aus diesem state benötigen, verbinden sich mit dem store, um bei jeder state-Änderung per Callback-Funktion `stateChanged(state)` darüber benachrichtigt zu werden und ggf. ihre properties upzudaten.
State-Änderungen können ausschließlich über vorher definierte actions (vgl. actions.ts) geschehen, die vom reducer (vgl. reducer.ts) auf den aktuellen state angewendet werden.

Damit nicht bei jeder kleinsten Änderung des states jedes Element upgedated werden muss, wird die Bibliothek reselect eingesetzt. Die mit ihr definierten selectors (vgl. selectors.ts) ändern ihren Wert (eine vom state abgeleitete Information) nur, wenn der entsprechend wichtige bzw. relevante Teil des states verändert wird.

Die Zeitreiheninformationen über die Werte des Simulators werden auch im state gespeichert und zwar so, dass sie von der eingestzten rendering engine, WaveDrom, leicht verstanden werden können. Dazu werden die Werte in String-Arrays gespeichert, allerdings wird statt einem wiederholten Wert ein `"."` gespeichert.

Da zusätzlich zu den Standard-Fähigkeiten von WaveDrom noch das Highlighten von h*-problematischen Eingangsbelegungen bzw. Zuständen, sowie das Rendern von vertikalen Linien zu Demonstrationszwecken gefordert war, kommen zusätzlich zum `signal` noch `hStar` und `vLines` hinzu. Diese machen dem Standard-WaveDrom-Rendering keine Probleme, d.h. man kann sie getrost im WaveJSON abspeichern und das Resultat kann immer noch von jeder WaveDrom-Anwendung gerendert werden (abzüglich h* und vLines). Für dieses Projekt wurde deshalb (und aus Kompatibilitätsgründen) der WaveDrom Code leicht angepasst.

Für die Kommunikation zwischen Visualizer und Simulator gibt es fünf CustomEvents: `InitializeVisualizerEvent`, `UserChangeEvent`, `SimulatorChangeEvent`, `ClockEvent` und `ResetEvent` mit den jeweiligen Event-Namen `'lv-initialize'`, `'lv-user-change'`, `'lv-simulator-change'`, `'lv-clock'` und `'lv-reset'`. Die ersten drei davon haben eine property namens `args`, die die jeweiligen Argumente zusammenfasst. Die Struktur dieser Objekte kann folgenden Interfacedefinitionen entnommen werden (? bedeutet optional):

```
interface InitArgs {
      inputs: Array<MachineVariable>;
      states?: Array<MachineVariable>;
      outputs: Array<MachineVariable>;
      canSetState: boolean;
}
interface UserChangeArgs {
      inputs?: Array<MachineVariable>;
      states?: Array<MachineVariable>;
}
```

```
interface SimulatorChangeArgs {
	states?: Array<MachineVariable>;
	outputs: Array<MachineVariable>;
	hStar?: string;
	answeringClock: boolean;
}
interface MachineVariable {
	name: string;
	value: string;
}
```

Die Daten für die Übersetzung werden jeweils in “defaultNS.json” gespeichert, in einem Ordner mit Namen des Sprachkürzels (z.B. “en” für Englisch) unter “locales” (also z.B. “src/app/locales/en/defaultNS.json”). Die Dateistruktur ist ein Objekt, dass key-value-pairs enthält (jeweils strings):

```
{
	"keyString": "is translated to this",
	...
}
```

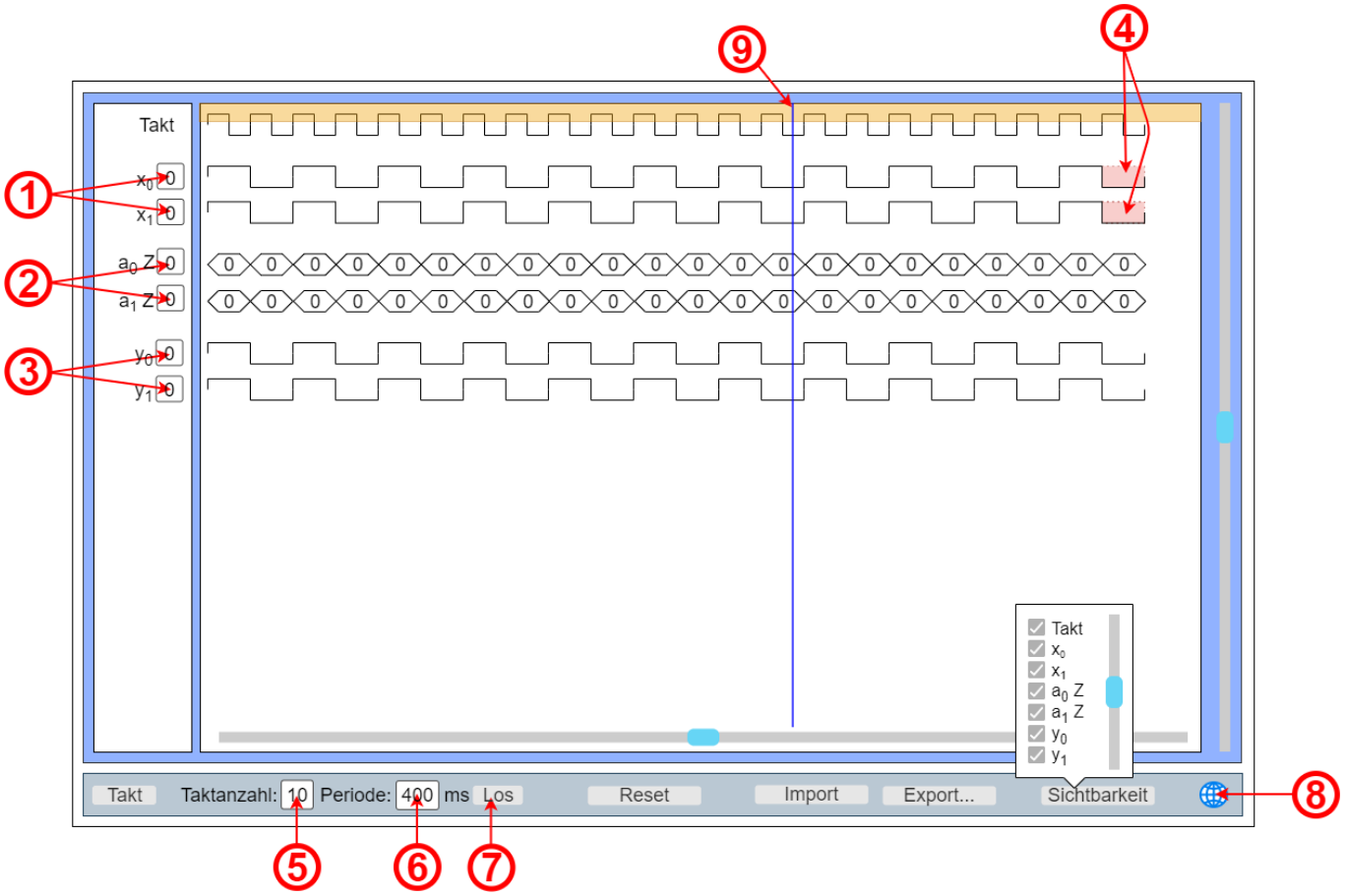
Dabei muss natürlich derselbe key für alle Sprachen benutzt werden. Wenn der translator von “translator.ts” importiert wurde, wird auf die Übersetzung wie folgt zugegriffen:

```
translator.t("keyString")
```

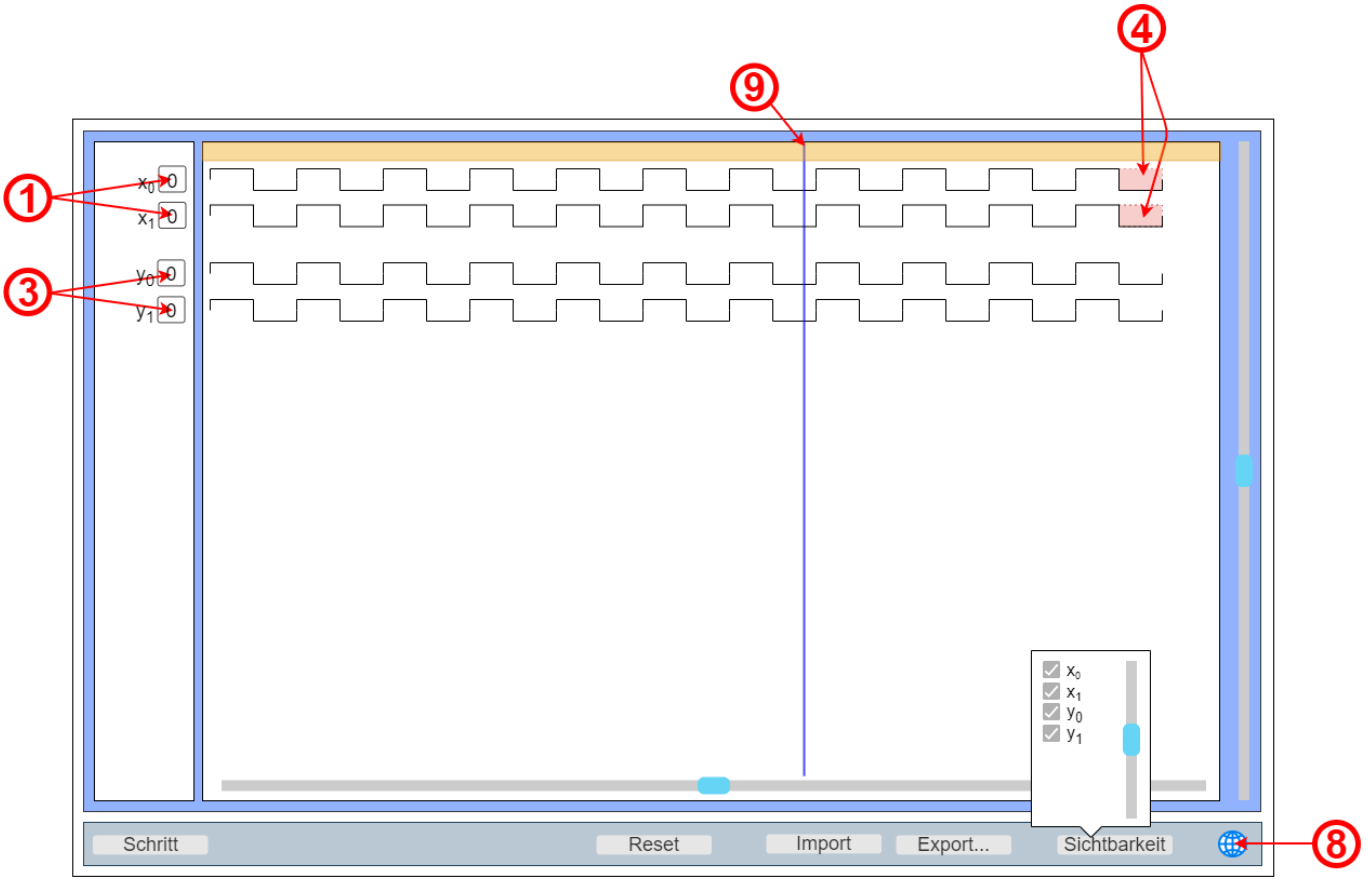
Dieser Aufruf würde, sofern das obige Beispiel in “locales/en/defaultNS.json” gespeichert und die Sprache Englisch ausgewählt ist, “is translated to this” zurückgeben.

5. Benutzeroberfläche

5.1. Mockup für sequentiell



5.2. Mockup für kombinatorisch



5.3. Erläuterungen

(1) dient sowohl dazu, den aktuellen Wert der Eingangsvariablen anzuzeigen, als auch als Button, um den Wert zu toggeln.

(2) dient hauptsächlich dazu, die aktuelle Zustandsnummer anzuzeigen. Sofern das Simulatorsystem dies unterstützt (siehe 3.), kann man hier auch vor Simulationsstart (bzw. nach einem Reset) eine Zustandsnummer eingeben.

(3) zeigt lediglich den aktuellen Wert der Ausgangsvariable an.

(4): Klickt man hier (aktueller Takt), so wird ebenfalls der Wert der Eingangsvariable getoggelt.

(5) dient zunächst als Eingabefeld, um festzulegen, wie viele Takte beim Klicken von **(7)** ausgeführt werden sollen. Nach dem Klicken von **(7)** wird hier angezeigt, wie viele Takte noch übrig sind (Bearbeiten nicht möglich). Wird **(7)** nochmal geklickt, bevor “0” erreicht wurde, wird die aktuelle Zahl übernommen und man kann sie wieder bearbeiten. Andernfalls wird wieder die ursprünglich eingestellte Taktzahl übernommen.

(6) dient als Eingabe für die Periodendauer der Takte während Mehr-Takt-Ausführung. Ähnlich zu **(5)** wird das Bearbeiten währenddessen blockiert.

(7) startet die Ausführung mehrerer Takte. Nachdem es geklickt wurde, ändert sich der Text zu “Stopp” und erneutes klicken von **(7)** bewirkt den Abbruch der Mehr-Takt-Ausführung.

(8) öffnet eine Sprachauswahl.

(9): Durch einen Doppelklick im orange markierten Bereich wird zu Demonstrationszwecken eine vertikale Linie eingefügt. Die Linien rasten immer auf ein Viertel eines Takts ein. Erneutes Doppelklicken auf die Linie löscht diese. Die Linie erstreckt sich über die gesamte Höhe des Diagramms.

Der Button Takt bzw. Schritt weist das Simulatorsystem an, einen einzelnen Berechnungsschritt zu machen und die neue Ausgangsbelegung und ggf. die Zustandsnummern zurückzuschicken.

Der Button Reset startet die Simulation neu.

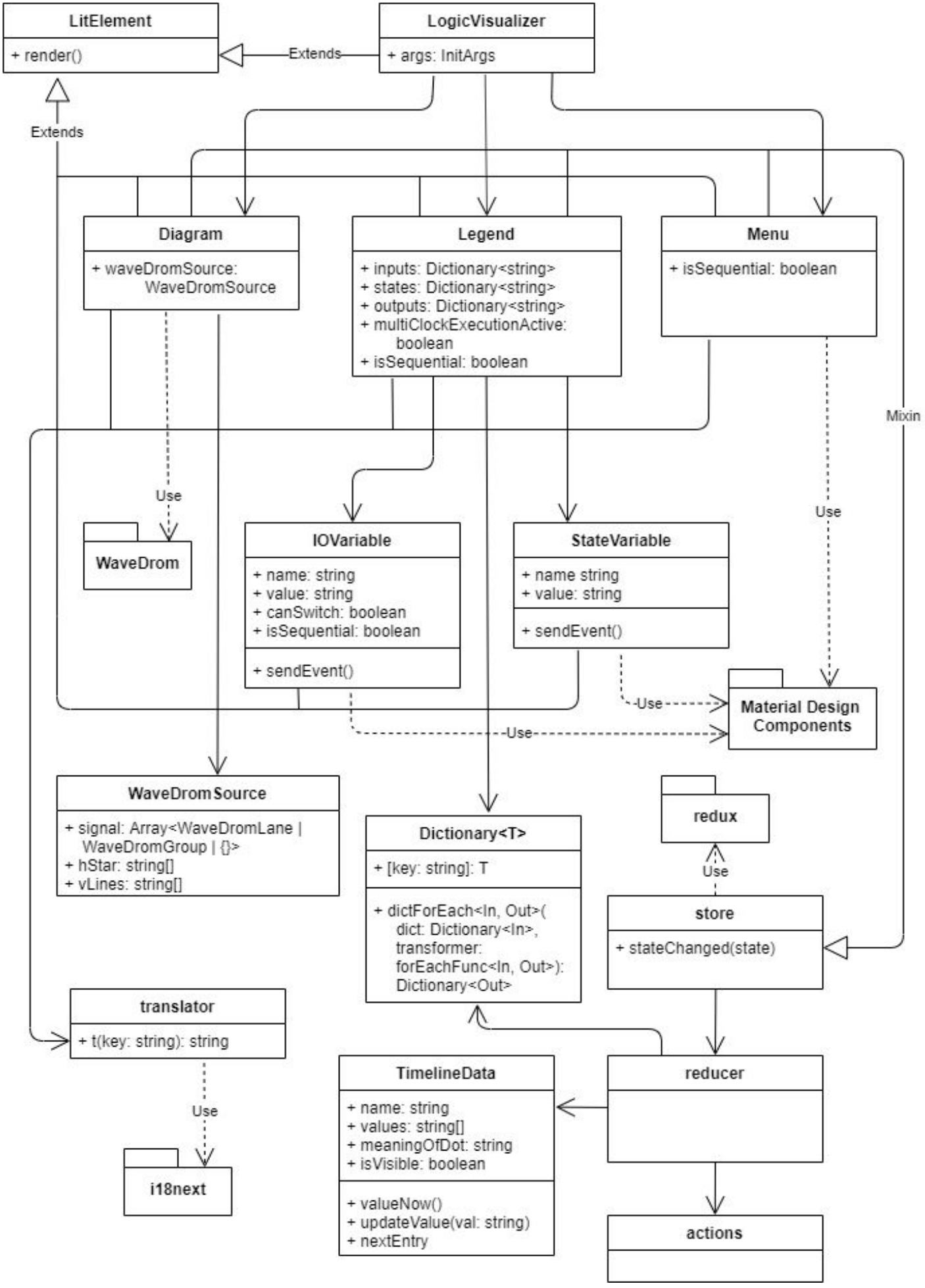
Der Button Import lässt den Nutzer eine Datei auswählen, die ein gespeichertes Diagramm im WaveJSON-Format zur Anzeige lädt. Dies kann unabhängig von einem Simulator geschehen, da die Simulation nicht fortgesetzt werden kann.

Der Button Export öffnet einen Dialog, in dem der Nutzer auswählt, in welchem Format (SVG, PNG, WaveJSON) er das Diagramm exportieren möchte. Nach einer Auswahl wird die entsprechende Datei auf seinem Gerät gespeichert.

Der Button Sichtbarkeit öffnet das dargestellte Menü, in dem man durch An- bzw. Abwählen der Checkboxen die Sichtbarkeit der einzelnen Diagrammzeilen steuern kann.

Die horizontale Scroll-Bar dient zum zeitlichen navigieren durch das Diagramm. Sie bewegt nur den weißen Bereich, in dem sie sich befindet.
Die vertikale Scroll-Bar bewegt alles, was im blauen Bereich ist, gleichmäßig.

6. Entwurf


LitElement
+ render()
Extends
LogicVisualizer
+ args: InitArgs
Extends
Diagram
+ waveDromSource: WaveDromSource
Legend
+ inputs: Dictionary<string>
+ states: Dictionary<string>
+ outputs: Dictionary<string>
+ multiClockExecutionActive: boolean
+ isSequential: boolean
Menu
+ isSequential: boolean
Mixin
Use
WaveDrom
IOVariable
+ name: string
+ value: string
+ canSwitch: boolean
+ isSequential: boolean
+ sendEvent()
StateVariable
+ name string
+ value: string
+ sendEvent()
Use
Use
Material Design Components
Use
WaveDromSource
+ signal: Array<WaveDromLane | WaveDromGroup | {}>
+ hStar: string[]
+ vLines: string[]
Dictionary<T>
+ [key: string]: T
+ dictForEach<In, Out>(dict: Dictionary<In>, transformer: forEachFunc<In, Out>): Dictionary<Out>
redux
Use
store
+ stateChanged(state)
translator
+ t(key: string): string
Use
i18next
TimelineData
+ name: string
+ values: string[]
+ meaningOfDot: string
+ isVisible: boolean
+ valueNow()
+ updateValue(val: string)
+ nextEntry
reducer
actions

7. Benutzte Software

Die wichtigsten Dependencies sind hier angeführt. Für eine vollständige Liste, siehe “package.json”.

7.1. Frameworks

- 7.1.1. Polymer LitElement 2.3.1: https://lit-element.polymer-project.org/
- 7.1.2. redux 4.0.5: https://redux.js.org/

7.2. Libraries

- 7.2.1. i18next 19.3.4: https://www.i18next.com/
- 7.2.2. Material Design Components 5.1.0: https://material.io/components/
- 7.2.3. reselect 4.0.0: https://github.com/reduxjs/reselect
- 7.2.4. WaveDrom: https://wavedrom.com/

7.3. Entwicklungstools

- 7.3.1. IDE: JetBrains WebStorm: https://www.jetbrains.com/webstorm/
- 7.3.2. Versionierung: Git
  - 7.3.2.1. GitLab der TU Ilmenau: https://gitlab.tu-ilmenau.de/
  - 7.3.2.2. GitHub Desktop: https://desktop.github.com/
- 7.3.3. Dokumentation: GoogleDocs: https://docs.google.com/
- 7.3.4. Zeichentool: draw.io: https://www.draw.io/

7.4. Sonstiges

- 7.4.1. GOLDi Labs: http://goldi-labs.net/

Entwickler: Andreas Gebhardt