Expertise
and insight
for the future

Thanh Nguyen

Statecharts for modern web application state management

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

6 March 2020

Author Title Number of Pages Date	Thanh Nguyen Statecharts for modern web application state management 54 pages 6 March 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of the school of ICT

Web applications are amounting to unprecedented relevance and importance as more and more businesses provide their products and services via the web platform. Because of the reactive and context-sensitive nature, the featureful web applications require intricate efforts of effective consuming and manipulating the application states (state management). While the proposition of an explicit, central representation of the application states from modern frameworks was a major step forward, the actual effort in modelling this representation remained largely unstructured and ad-hoc. The consequence is compounding difficulty to reason and implement effective state management as complexity grows.

Finite state machine is a rigorous mathematical model invented to tackle the challenges in reasoning about the transformation of states. Amongst many benefits it provides are the ability to have a diagrammatic view on most of the possible values via State Transition Diagram, the relation of those possible values, and the evolution path (transition graph). The graph nature of the transition diagram enables the analysis by the use algorithms and serves as an effective documentary, which improves the long-term maintenance of the software. For this reason, finite state machine has been the core enabler for applications of various scales, complexity, and criticalities, from spaceship control system to artificial intelligence in games, and now web applications.

While being appropriate for web applications, the sheer complexity typically cripples the benefits of Finite state machine and state transition diagrams by a phenomenon called state explosion. Statecharts is a formalism built on top of the former two in order to retain the benefits while avoiding state explosion.

The thesis introduces the ins and outs of Statecharts and its foundation – Finite state machines and State transition diagrams. It then demonstrates the concrete steps in how to utilize Statecharts in a modern case study web application: a local music player. In more details, it elaborates on the modelling of the central representation of the application states, designing the manipulation and consumption of this model, and the integration with other frameworks. The final section evaluates the claims on the benefits of Statecharts over the popular ad-hoc approach.

Keywords	Statecharts, Finite state machines, State transition diagram, Xstate, Javascript, Vue, Xstate

Contents

List of Abbreviations

FSM	Finite state machine. A mathematical model of computation revolving around a behavior of a system as it undergoes through different states.
DOM	Document object model. is an application programming interface (API) for valid HTML and well-formed XML documents.
API	Application Programming Interface. The interface or communication protocol that different programs or different parts of the program relies on in order to communicate or influence each other.
STD	State transition diagram. A visual representation of finite state machine were states are inputs are denoted as a directed graph.
HTML	Hypertext Markup Language. The standard markup language for documents designed to be displayed in a web browser.
I/O	Input/Output. Usually refers to the communication between an information processing system with the outside worlds, such as writing or reading data from network or disk.
CSS	Cascading Style Sheet. This a language that describes the style of an HTML document.

1 Introduction

Web applications as of today are expected to be increasingly featureful, as an accelerating number of businesses are looking to be empowered by web technologies to deliver their products and services. Because of such interest, web applications have evolved from being relatively static to highly dynamic, reactive and thus complex software. The typical web application is expected to be highly interactive, from reacting to the user inputs and changing the interface responsively, scheduling animation, communicating with the backend systems to retrieve data, to even video and audio manipulation.

Although many challenges regarding building the graphical user-interface has been met with quality solutions like React, Angular or Vue and a plethora of accompanied technologies, one lingering pain to a large portion of developers is managing the application state correctly. The challenges are, to a large extend, due to the reactive and context-sensitive nature of user interface. As users interact with the interface, they generate events that evolve the conditions of the software. This then drive further alterations and reactions, however dependent on past interactions (the context). For example, a click can trigger a complex interaction with the backend system, which in turn triggers a variety of different reactions varying by what the user has done before, the settings, or the responses from the backend system, etc.

As a remedy, the modern web frameworks suggest and popularize a centralized a data model that served as a representation of the context of the application. While being a major step forward in bringing more visibility on the application state, the efforts of actual modelling such model are largely arbitrary. Consequently, they rarely accurately represent the real states, their relations, or possible evolution paths, in way that can be easily reasoned about. Because the application is driven by this representation, a poor attempt most likely results in great difficulty in manipulation. Consequently, a decision made on mismatched state representations introduce insidious bugs of all kinds, from unwanted network calls to silly glitches of having both error message and spinner on the screen, at the same time.

State machine is a well-studied construct employed in reactive applications of all scales, criticalities and complexities. Some well-known example could be cited, such as Nasa's

Space Launch System Solid Rocket Boosters [1], AT&T's Email agent controller [2], etc. Other highly complex applications with a user interface, namely games, are also empowered by state machines, even to the degree of artificial intelligence [3]. Inspired from the gaming and embedded world, web application state management may greatly benefit from state machines as well.

The goal of this thesis is therefore to explore state machine, as well as an improved formalism of such called statecharts. In doing so, the paper discovers how statecharts can be of help for complex state management in web applications. Concretely, it aims to introduce the features of Statecharts and a library that implements executable statecharts in the browser named Xstate. There follows a case-study implementation of a simple music player, where the author demonstrates and evaluates state management using the Statecharts and Xstate.

2 Current state analysis

2.1 User interface as a function of state

Popular web frameworks such as React, Vue, Angular, Svelte, and many more all assumes and advocates for one important idea: the user interface is a projection of the underlining application state [4]. In this philosophy, web developers construct a body of code, called component, that computes a description of the user interface given a set of input, which is the application state. The result is consumed by the framework, transforming the description into real user interface.

This is a great idea since it cleanly draws the line between two distinct tasks: user interface development and state management. Classical development of markup, style and display logic happens in the construction of these user interface units, known as components. At times, components might internally generate and consume their own local data, however it is generally still correct to think that the user interface is mapped from model. The other task, which is state management, became surprisingly challenging to developers the more complex the application becomes. [5]

The typical default approach to model the representation of the application state is either via domain modelling, or ad hoc. Consider a typical network request scenario that is triggered by a button click fetching a list of to-do items.

```
type IState = {
    loading: boolean;
    todos: Array<string>;
    errorMessage: string | null;
}
```

Listing 1. Type declaration of a piece of state keeping track of fetching some to-do items from the backend, in Typescript.

In the approach above, three fields are used to keep track of the network state while some of which are overloaded for stateful data, namely the message error and the to-dos. At first glance, this representation satisfies the immediate need. However, a deeper analysis reveals concerning problems.

It is intuitive to observe that there are four states at play: initial, loading, success and failure in networking. This stems from the natural states of a Javascript Promise (uncreated, pending, resolved, rejected) that typically represents a future value, such as that of network response. These four states are mutually exclusive, and there is a strict sequence that these states may transform from one to another. The model above does not reflect this relationship of exclusion and transition. The constrains in updating the representation exists, however those are implicit and dependent on non-obvious checks in the manipulating code.

Secondly, since variables such as **errorMessage** and **todos** are overloaded for conveying both the state of the application and the data, the natural tendency is to conveniently introduce data transformation logic within the same state manipulation code. This usually leads to additional cognitive overhead that may result in mistake due to forgetfulness. In the scenario of a successful request, transforming the payload of to-dos received from the server may increase in the updating function complexity, rendering the developer more prone to forget resetting the value of the error message and the loading flag.

Thirdly, not only this is a poor representation, which is error-prone to manipulate, it is difficult to consume as well. Since a set of known states are poorly defined, as well as their relationship with each other, invalid combinations of conditions might occur. This might allow for corresponding forbidden actions to happen when they should not happen. The author's experience with code bases employing this strategy ranges from slightly annoying bugs to serious data corruptions, i.e. when some user attempted to click the button repeatedly while the network request is inflight, leading to multiple requests made due to button not disabled, residual error message displayed, race conditions in requests, etc.

The example given is contrived enough yet sufficiently problematic to cause real production problems. Enhancing the application with additional features, such as timeout, retry, polling, chaining requests, etc. using the same design drastically increases the application complexity that renders maintenance costly and delivers unpleasant experience to both developers and users.

3 Theoretical background

Web application, or any user interface in general, is context sensitive. The user interface is driven by a what has happened before, different effects and different screens are outputted, despite similar events from the user. For example, in a music player application, a click to the very same play/pause button may sometimes pause or play or do nothing at all. This is dependent the status of whether something is playing, or there is anything to be played at all. [6]

Thinking in terms of events and states make sense to adopt an approach where it is clear how the context influences the application, due to what set of possible events, and how the changes will transpire instead of a structure that does not inform and enforce such constraints. Fortunately, there exists a well-known construct that models such context-sensitive applications excellently, and that is finite state machine. [7, p. 29-30]

3.1 Finite state machine

Finite state machine (FSM) or finite-state automata (FSA) is a mathematical model of computation. In a finite state machine, there is only a countable number of states (S). A state is a set of stable conditions in which the machine waits for more inputs to be supplied. At any given time, the system can only be in one state. For example, in a light bulb system there are two states, namely on and off. The light bulb cannot be both on and off at the same time. [6]

An example FSM moves from one state to another by accepting a finite sequence of input. When modelled for a reactive system, such as that of web applications, the inputs are typically recognizable events. For instance, a response from the server.

The process of leaving a state to enter another is called a transition. Every accepted input causes a transition, but not always to a new state. At times, an input may cause a transition that goes back to the same state. This is called self-transition. Actions might be carried out upon the transitions taking place. In the aforementioned light bulb example, the event **TURN_ON** is accepted input in the light bulb system causes the transition to the **on** state and carry out an action of allowing the electricity current to go through the filament.

A finite state machine is deterministic if and only if the next state is only dependent on the value of the input and the value of the current state. There is no higher context or hidden dependencies. There exists non-deterministic finite state machine where the relationship of next state does not only depend on the current state and the input, but this is of little to no interest due to the importance of determinism software system requires. In short, a FSM can be thought of as a set of functions in the following form: [6]

$$Current\ state, input \rightarrow output, next\ state$$

Finite state machines are typically represented using a notation. One very useful and typical notation is the state transition diagram. [6]

3.2 State transition diagram

State transition diagram (STD) is a visual representation of a finite state machine. It was invented during the 1950s at Bell's research lab for the control mechanism of telephone line switching [6]. The visual medium is useful due to its descriptive yet rigorous nature.

STD is simple as it only has to elements: nodes and labelled arrow representing states and transitions, respectively. Arrows originates from a node and the head points to the target node. These forms a directed graph [6]. For example, consider an abstract FSM represented by STD in Figure 1.

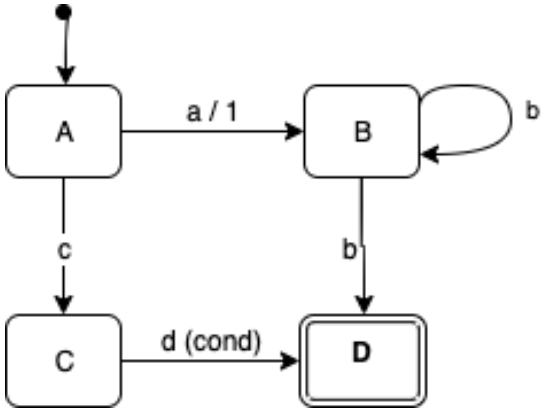

Figure 1. A state transition diagram with four states

In figure 1, the FSM has four states: **A**, **B**, **C** and **D**. The state **A** has an arrow originating from a black thick dot pointing to it. This is a marker to convey that **A** is the initial state, which means the machine starts in therewithin. The state **D** has a special box denoting that it is a final state. The machine terminates and stops responding to any more inputs upon reaching any of the final states. [7, p. 63-64]

The state **A** has two transitions to both states **B** and **C** caused by events **a** and **c**, respectively. An action may be specified by appending the action name after the event name, separated by the slash. When the machine takes the transition from A to B, the action 1 is executed. A transition may not always lead to another state, as can be observed in the self-transition of event b in state B. [6]

State transition diagram stood the test of time until today as it provides the explicit description of the application behavior. The identification of the states and events results in as an easy task in identifying design flaws, problematic interactions, as well as non-obvious outlining circumstances [8 p. 380]. Secondly, a visual medium allows ease of understand and reasons about why and when the application behave in a way. This results in a system that can be largely understood top-down without examining the code, which may be beneficial to user experience designer, product owners, and others. Thirdly and finally, there exists well-established techniques to formally verify the correctness of the modelled system, ensuring certain quality of safety and liveness [9].

3.3 State explosion

STD is a powerful technique to model the system that brings compelling benefits. Unfortunately, under sufficient complexity, it breaks down due to a phenomenon known as state explosion. [6 p. 48, 8 p. 380]

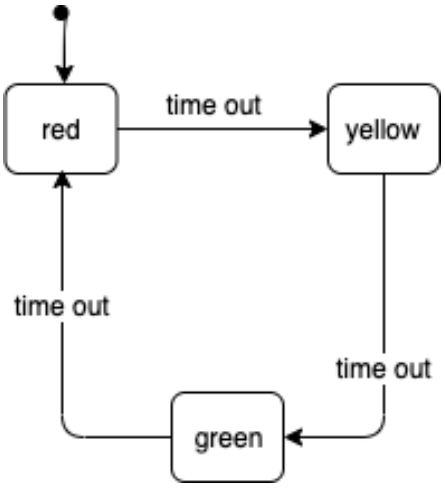

Figure 2. A Simple STD for a traffic light machine

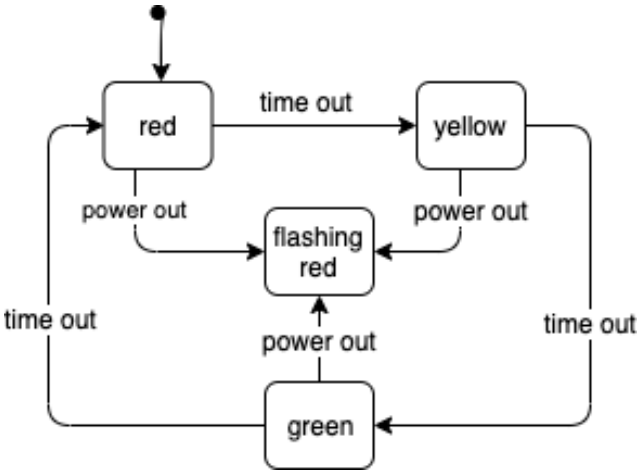

Figure 3. An exploded STD for a traffic light system, adding a flashing red state.

Consider a case where the traffic light system, as seen in figure 2, is enhanced with a new feature. In addition to a **normal** operating mode of three colored lights is an emergency mode where the red light rapidly flashes. The figure 3 demonstrates the disproportioned growth in the number of transitions resulted from this minor extension. Because STD can only express states in a flat fashion, it is not possible to represent the collective **normal** state, hence the behavior of “go from normal mode to flashing red mode” must be conveyed by on the **red**, **green** and **yellow** states.

Modularity plays a vital role in software development. Within any application, there certainly exists multiple modules, each have their own states. For example, consider a rich text editor application. The state of how the typographic formatting, as seen in figure 5 and text alignment, as in in figure 4, are independent of each other.

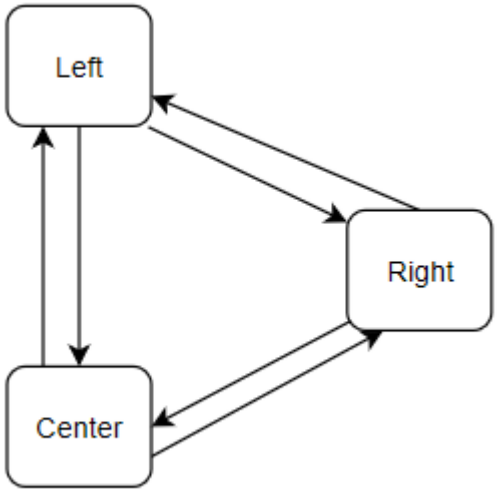

Figure 4. STD for typographic alignments. Transition arrows label has been omitted for brevity.

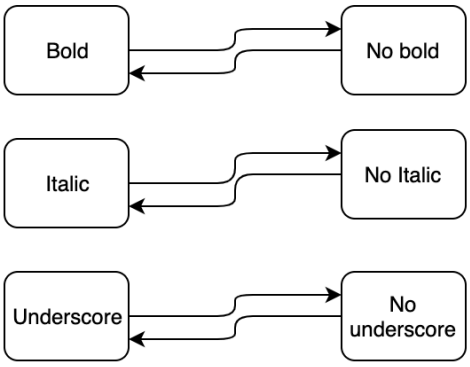

Figure 5. A STD for typographic formatting. Transition arrows label has been omitted as well

Figure 6 illustrates that such a diagram to represent both modules requires every and all combination be explicitly stated. Even at the simplest of scale, the diagram of 24 states, which is the product of the number of states; devolves beyond legibility. This is another aspect of state explosion: with a modest amount of independent states introduced, the number of states multiplies.

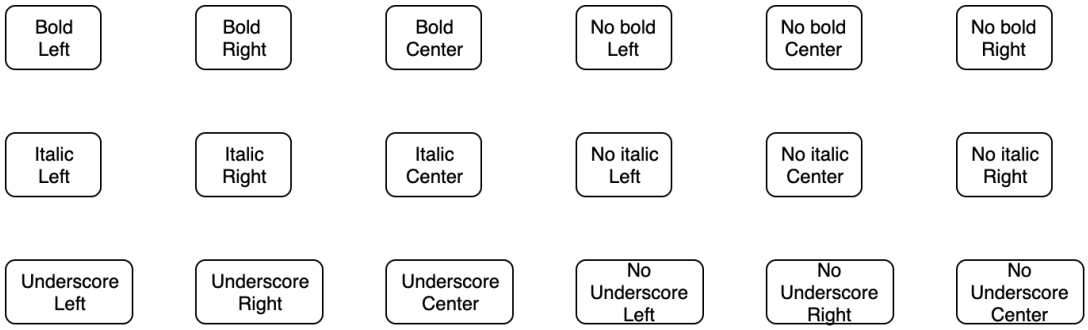

Figure 6. Word processing machine STD. Transition arrows are omitted for legibility.

The conclusion that can be drawn is that while STD is appropriate to for modelling reactive applications the likes of web user interface, any app worth modelling is already complex enough to presents insurmountably illegible STD due to state explosion. While it can be understood that state transition diagram is yet another cautionary tale of failed innovation [10], all is not lost if there exists a formalism that can express the hierarchy as well as concurrency of states and allows for conditional transitions. Thankfully, in 1987, an Israeli computer scientist came up with another graphical formalism that possess such improvements: Statecharts, invented by David Harel.

3.4 Statecharts

Statecharts was invented while David Harel was commissioned to design an aircraft navigation system in 1987 [11]. Statecharts is a visual formalism built on top of state transition diagram but specifically with the purpose of addressing the state explosion phenomena. Statecharts introduces the concepts of hierarchy, concurrency, conditional transitions with guard, broadcast communication, history states and many other constructs thereby allowing complex states to be decomposed, understood and implemented. This brings unprecedented expressive power that allows brevity yet powerful, precise, and scalable specification for large systems, even for web applications.

3.4.1 Hierarchy of states

Contrary to vanilla STD, where all states exist at the same level, states in statecharts can contain sub-states, or be contained in a super-state. This forms a hierarchy of states which may be used to reflect different levels of details, or abstractions.

The hierarchy of states is expressed using encapsulation. Rounded rectangle is used to denote states at any level. There is an appropriate default states at each depth. A transition into a super-state is also considered to be a transition into the default sub-state of that super state, so on and so forth down the hierarchy. Likewise, the system leaves all of the sub-states when the super-state is transitioned away.

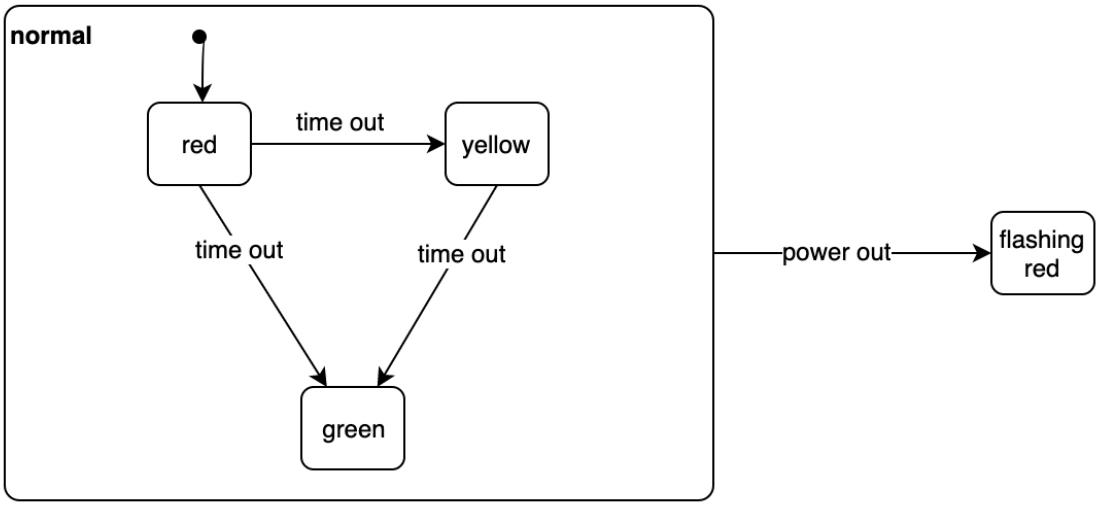

Figure 7. A Statecharts for traffic light system

The Statecharts version of the traffic light system in figure 7 reveals two important differences compared to the one shown in figure 3. The diagram introduces a rectangle labelled as **normal**, enclosing the familiar three **red**, **green** and **yellow** states. This change is sometimes called clustering as states are grouped underneath a super-state. For this example, the three colored states are clustered into the **normal** state. As a result, the diagram now better reflects the “operating normally” state in the intuition of “the traffic lights switch between each light while in normal operation”.

Secondly, the transitions from the three states to the **flashingRed** state was consolidated into a single transition from their super-state. When the system transitions to

flashingRed state, it exits the color states as well. Inversely, when the streetlight system transits in the opposite direction, from **normal** to **flashingRed**, it takes the transition into the child-state **red** at the same time.

By introducing a new composite state and thereby raising the depth of the statecharts, the number of transition arrows drastically decreased. In fact, if there exists an alternative streetlight system of sixteen million colors, there would still be a single transition to **flashingRed** state, not sixteen million. This property explains why statecharts can scale to complex system without suffering from the proliferation of arrow transitions. The hierarchy allows factoring out common transition origins and targets.

Furthermore, the hierarchy of states allows for abstraction and information hiding. For example, the three colored states may be omitted, or zoomed-out when the tasks concern only the relationship of the normal operating states and other special or abnormal states. One can design and reason the system on the more abstract behaviors, refining as one delves deeper in details and particularities. On the other hand, should the question regarding the normal behavior arises, Statecharts allow focusing into the **normal** state to work with the appropriate level of details. This allows the model to be useful for different individuals working at different levels of abstractions, from the grand system architect to a component developer.

As seen, the hierarchy of Statecharts allows for clustering and refining mode of working. Clustering states is the way of working in which common transitions are identified and factored out in the form of a super state. The developing of the model from figure 3 to figure 6 is such a case. Refining is, in contrary, the way of working in which a state is refined to contains substates, adding details. Both are common approach to software developments, and the fact that Statecharts supports both ways serves as a testimony to its appropriateness and strength.

In short, the hierarchy of states allows for reusability of transitions, simplifying the diagram significantly. It also represents states at different level of abstractions, which enables both top-down approach or clustering exist states in refactoring work. It makes possible a bird-eye views level of understanding, while still maintain sufficient particularities of any module of the systems.

3.4.2 Orthogonal state

As discussed, the accelerated multiplication in the number of states as those independent of each other are introduced is the second major ailment. The inflexibility of the vanilla STD begs for representing every explicit combination of this type of states, rendering the diagram chaotic and crippling any serious efforts to model sufficiently complex systems.

In Statecharts, independent states are are denoted by states next to each other, separated by a dashed line. The super-state containing the orthogonal parts represents the product of each of the state of the parts. The combination is implicit in exchange for brevity, the same way transition into sub-states is implied by transition into super-state. When the system transition to a state containing orthogonal regions, it simultaneously enters all the default states of each orthogonal states. Since these states are independent, there exists no transition between them. Orthogonal states are not limited to a root-level depth of Statecharts. Any state node can contain orthogonal sub-states.

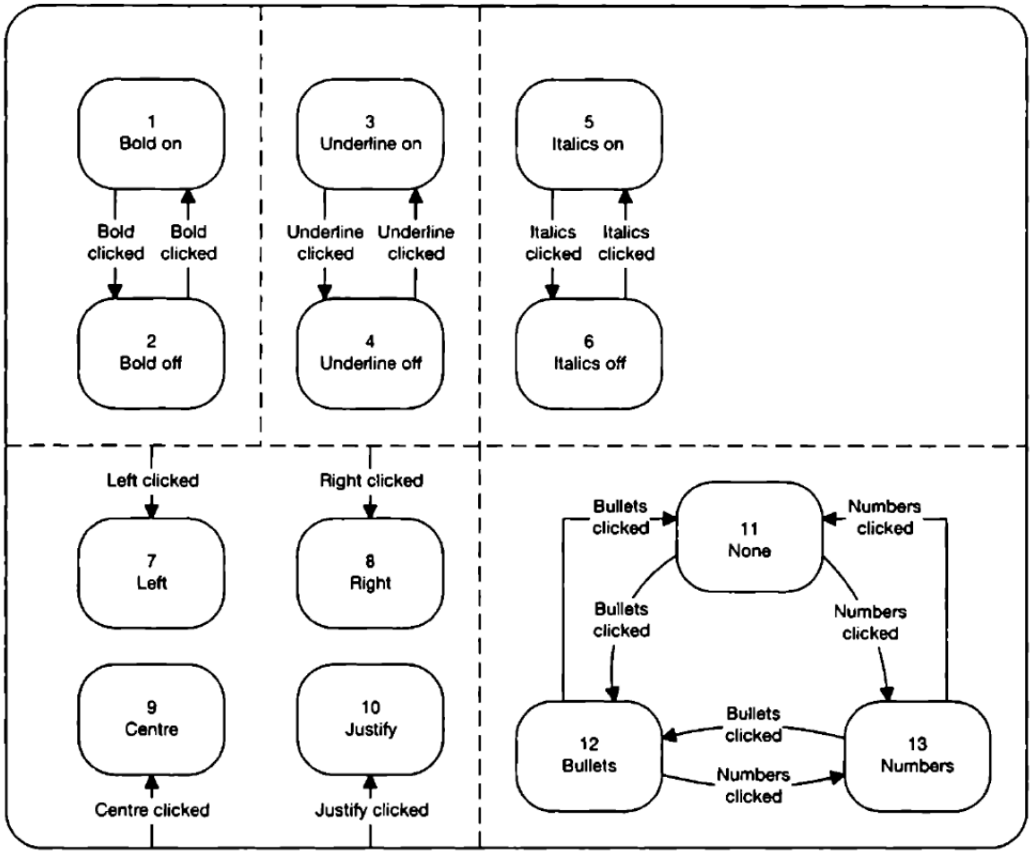

Figure 8. Statecharts for an advanced rich text editor

Figure 8 shows a Statecharts diagram for the text editor from figure 4 to 6, albeit with extra features. It is divided into five regions by dashed lines. These regions denote states that are parallel to each other, at the same depth. In a normal STD, this text editor would have had a total of 72 states somehow resulted from several ambiguous (non-explicit states).

Orthogonality finds its application in expressing modularity, because any system typically includes multiple modules that do not necessarily depend on each other. Orthogonality in Statecharts prevents the situation where every combination of independent states must be represented, thereby mitigates state explosion. Hierarchical and orthogonal states provide the explicitness and brevity needed to sufficiently model the control layer of web applications.

3.4.3 Actions and activities

So far, the FSM represented by the statecharts can only react to events in changing its internal states. However, real application requires influencing the environment. The machine must be capable of executing effects, such as establishing network requests to retrieve remote data, performing I/O operations, rendering to a screen, or emitting more events. This points to a need for Statecharts to specify the influence the FSM may exert to other components and systems by means of generating events and setting values.

Statecharts defines the output of the FSM into two categories: actions and activities. Actions are effects which happen instantaneously, usually in a fire-and-forget fashion, such as setting some values, logging into the console, sending an event to itself (broadcastting), etc. Activities, on the contrary, are effects that take a non-zero amount of time to happens. This could be establishing a network connection to fetch remote data or running a timer.

In Statecharts, effects can happen at three points: during transition, upon entering a state, and upon exiting a state. The order of execution is that exit actions of the origin state happens first, followed by transition actions, and finally entry actions on the target state.

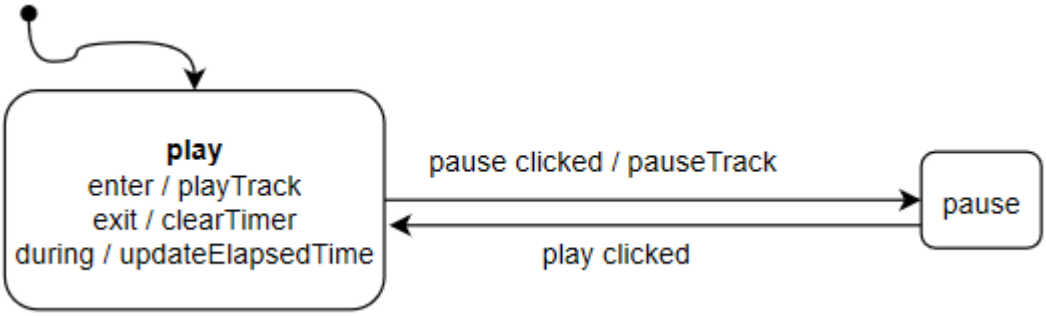

Figure 9. A Statecharts of a music play back machine where the actions are specified

Consider the statecharts of a simple music player composed of two states: **play** and **paused** in figure 9. There exists four actions: **pauseTrack**, **playTrack**, **updateElapsedTime**, and **clearTimer**. When the machine undergoes a transition from **pause to play** state, the following happens:

- First, all exit actions of the **pause** state are executed. There is none specified, however.
- Then all transition actions, which there is none as well, are executed.
- Following that, all entry actions of the **play** state are executed. This means **playTrack**.
- Finally, the activities bound to the **play** state are executed.

In a super-state, internal transitions between its children states will not cause its entry and exit effect to be executed. However, if a super-state is transitioned away, all current sub-states' exit effect will take place. Likewise, upon entry, the sub-state that is entered will have its entry effect executed.

Activities are never associated with transition, due to the instant nature. They are typically bound to a state and are represented with keyword "throughout" or "during". Activities are often implemented in practice by a combination of entry and exit actions that start and perform cleanup the activity, like initializing a media stream and closing such stream. However, it is useful still to explicitly draw the distinction between the two.

3.4.4 Guard and conditional state

There are often cases where a target state may not be determinable solely on the event data and the current alone [7, p. 40-41]. One example, as seen in figure 10, could be the case of a search form. The state can only transition to **loading** from **waiting for input** if and only if the query string is non-empty. This condition in a transition is called guards. Guards can be specified in the label of the transition arrow, in parentheses, next to the event name. In this example, two guards are **has query** and **query is empty.**

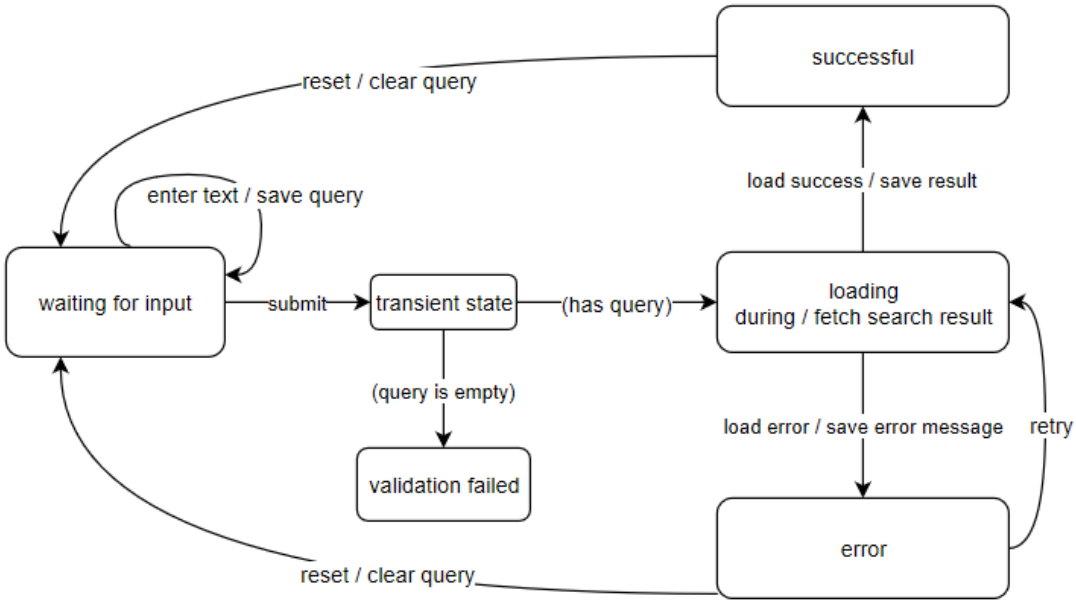

Figure 10. A statecharts for a search form with a transient state and guards.

Continuing with the form example, what happens if instead of forbidding the transition, an error message would be showed. A new state, **validation failed**, would be required. The ardent readers will notice that the form submission event play no part in determining whether the next state would be **loading** or **validation failed**. In fact, the event only causes the transition away from **waiting for input** state. It is the query string length being non-empty that determines the next state. Facing this dilemma, transient state could be leveraged.

Transient states instantly transition away without any associated events. That means the state will immediately exits upon entering, hence transient. Transient state bears another alias: pseudo state, because it is not a stable condition where the machine can wait for any input. Transient states may have entry and exit actions, but no activities.

Transient state is typically used in combination with non-overlapping conditions so that one path can be taken at a time, effectively making the transient state a router where the path can be determined by guards. In our example, a transient state would be used to decide if the next state would be **validation failed** or **loading**.

3.5 State machines actor

The orthogonal states are the main answer of Statecharts to the challenge of modularity, as discussed. However, likely are at least some form of dynamic modules may show up in the application at run time. For example, consider simple social media application. Each of the friends of the user is a module that has some states, such as online/offline, notification sounds, loading of information, etc. It is impossible to know how many and what specific friends a user have in advance. The question one may ask, is how these dynamic modules can be modelled using statecharts.

One might argue that only a central supervisory state is necessary. States per friend module can be manipulated as pure data instead. For example, setting the online state to on/off and using some sort of flags to keep track of network requests in loading friends' data. This bears great resemblance to the approached discussed in the current state analysis section. The arguments apply: if the features are trivial, such as simple data fetching and displaying online status, perhaps this is tolerable. That may not prove so manageable given sufficient complexity.

One solution to this problem is considering each of the friends a separate machine. Since Statecharts machine can both receive and output events, the main application can interface with these "friends-module". This pattern is inspired from the Actor Model, which is another topic on its own, thus this thesis will only attempt to give a brief overview.

The actor model essentially states that in a system, there can be different unit of computations that response to messages and carries out computation instead of direct calls. This plays well with the intuition that the business logic may be delegated to different state machines that talks to each other. In fact, in some case, the implementation of an actor is a finite state machine [12]. Typically, actors are formed in a hierarchy. Parent actors may create and terminates children actors, so on and so forth.

Independent, dynamic modules benefit greatly from an actor state machine implementation. Not only the event/message communications already familiar, different actors may be distributed to different execution contexts, i.e different threads. This approach forms powerful composability allowing complex behaviors and concurrent processes to be modelled and understood. For example, independent modules that operates in web workers, cross-boundary distributed simulation in multiplayer gaming, etc.

3.6 The benefits of Statecharts in web applications

With hindsight, most of the impediments in managing states are born from the difficulty to reason about the changes in the states. The prime contributor to the dilemma is a subpar model that is opaque about the way the states evolve and the relation between them, followed by convoluted manipulation code and ad-hoc effectful procedures. The popular approach to model the state of a network request machine demonstrated in section 2 makes it is visible how the exclusion of states is obscured whether there exists an enumerable number of states, amongst many other ills.

Finite state machine has all the answers to these thorns. With state transition diagram, the complete set of possible states are made extra clear via the graphical notation. The concept of transition illustrates the way states evolve as a trajectory that can be observed and reasoned about with graph algorithms. If traditional logging provides a time-travel (looking into the past) knowledge of how the state evolved, the diagrammatic graph of transition provides a foundation for debugging in the future as well. This is, in fact, done using temporal logic in the domain of model checking for finite state machine. Moreover, finite state machine empowers developer to identify the relation of the states in the system. Exclusion, hierarchy, independence, and conditionality are crucial relations that the state model must contains in order to accurately reflect the real conditions of the

applications. Finally, the framework of actions and output events offers a powerful tool to answer for the difficulty in timing or scheduling computations.

Finite state machine is an appropriate method of modelling the representation of states in applications that are context-sensitive and reactive, which web application are. However, due to the ever-increasing complexity, state explosion poses unsurmountable challenges to the application of FSM in web application. Statecharts empowers this method of modelling by retaining the good parts while solving state explosion.

In short, applications whose central representation of the application states modelled with Statecharts can expect to enjoy greater ease in managing states. The graphical representation lends excellently with educational and documentational benefits. The graph nature empowers more serious, critical application to be verified and proved as well. Perhaps, finally, managing state may start becoming enjoyable to developers at last.

4 Method and material

4.1 Case study motivation

The second goal of this thesis is to demonstrate how Statecharts can be used in a real-world web application. The case study application will be a local music player, modelled and specified with statecharts. Furthermore, a feature enhancement involving coordination of concurrent network requests will be examined. This section of the thesis includes a brief introduction of the relevant technologies used in the development of the application.

4.2 Technologies and tools

4.2.1 Xstate

Statecharts is strictly a visual formalism to represent finite state machines, which says little about the underlining executable code. Typically, translating the model into code is

part of the development work. Statecharts may be implemented as a singleton class, or a more free-flowing function so long as behavior regarding transition, guards and actions are satisfied. However, this process is not particularly of productive nature and risks incurring of programmatic faults. Therefore, if there exists a solution to translate a model into executable codes, then it is sensible to employ it. Xstate is one such solution, and it is specifically designed for web applications.

Xstate is written in Typescript, which is a statically typed favor of Javascript. It is invented by **David Khourshid,** a Microsoft developer. In Xstate, the statecharts structure is encoded in a Javascript object that closely models the W3C's SCXML (State charts XML) specification [13]. Xstate also provides a diagram generator from the Javascript structure and this can be used as either a discoverability playground, or an evergreen description. Figure 11 (generated by XViz) demonstrates the Xstate structure equivalent to the statecharts in figure 7.

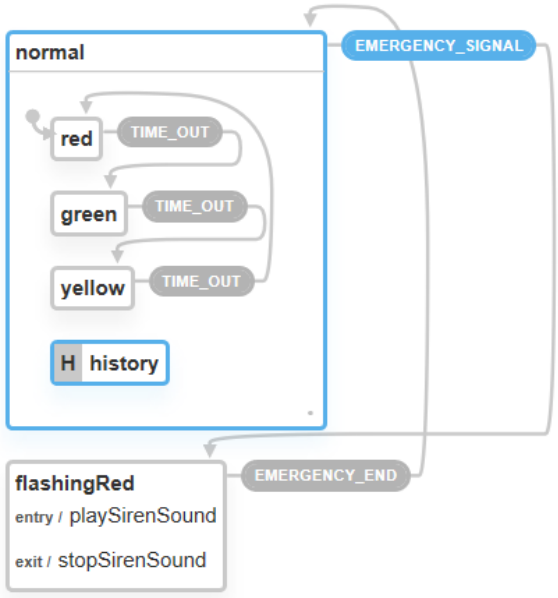

Figure 11. Interactive Statecharts diagram generated by Xstate visualizer tool.

```
const trafficLightMachine = Machine({
    id: 'trafficLight',
    initial: 'normal',
    states: {
      normal: {
        initial: 'history',
        on: {
```

```
        EMERGENCY_SIGNAL: {
          target: 'flashingRed',
        }
      },
      states: {
        red: {
          on: {
            TIME_OUT: 'green'
          }
        },
        green: {
          on: {
            TIME_OUT: 'yellow'
          }
        },
        yellow: {
          on: {
            TIME_OUT: 'red'
          }
        },
        history: {
          type: 'history',
          history: 'shallow',
          target: 'red'
        }
      }
    },
    flashingRed: {
      entry: 'playSirenSound',
      exit: 'stopSirenSound',
      on: {
        EMERGENCY_END: {
          target: 'normal',
        }
      }
    }
  }
});
```

Listing 2. A traffic light system Statechart specified in Xstate.

Xstate implements all the majority of the features of Harrell's statecharts. The framework provides supports for hierarchical, orthogonal, historic states, as well as guards, actions

and activities. Xstate provides basic mechanism for state machines as actors, including the parent/child relationship between machines. Finally, the framework provides observability to the current state values and the associated stateful data, termed context, for the convenience of the view layer, or any other consumers.

4.2.2 Vue

Vue is a progressive web web framework. In Vue, developers declaratively define composable units of the interface called components. A component composes of three parts: a script, a template, and an optional styling.

- The template is a superset of HTML, but with special syntax to interpolate values from Javascript, or to define rules on how the structure can be manipulated. It can include other components in the form of a custom HTML tag.
- The script defines the structure of the components. This includes values and methods the template consumes, as well as any additional data dependencies.
- The style provides a region where CSS styles can be defined for convenience.

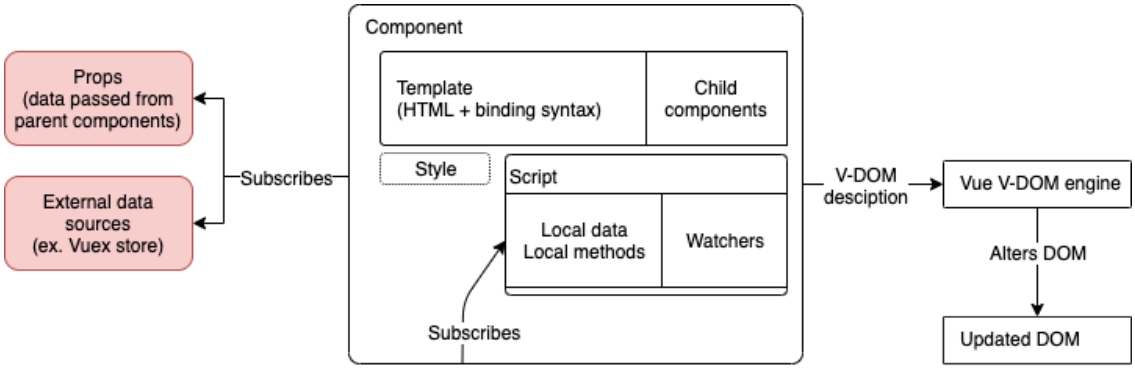

Figure 12. Structure of Vue

Vue subscribes to the philosophies of the user interface as a result of a computation. Each component is a function of the input data, mapping these to a structure that will be processed further by the Vue rendering engine and ultimately become what is seen on the screen. Figure 12 illustrates this relationship, as well as the constituencies of a component.

4.2.3 Vuex

Application states and shared data are a type of data that is meant to be consumed application-wise, not specific to any components. It is necessary that Vue can observe the value and be notified of changes in these data. It is possible to create a structure satisfying that requirement by using Vuex.

Vuex is a flux-inspired state management framework. It is typically used with Vue. With Vuex, one can define an application store, where globally shared data, such as the application state, can reside. Once configured, Vue components can subscribe to the store and be notified when changes happen, so that the UI never become stale.

Updating the view layer is typically one of the main tasks of the control layer. However, by leveraging the reactivity introduced by Vue and Vuex, this task can be significantly simplified. An executable Statechart built with Xstate can be contained in the store, at which point components can directly query the configuration of the state machine. This is less error prone and tedious than copying the current state value and update it in a superfluous variable.

4.2.4 Infrastructural tooling

In this case study, a number of other technologies are employed for infrastructural purposes. Specifically, three technologies are worth mentioning:

- Webpack: a toolchain to transform application code, for the purpose of bundling the code for deployment.
- VueCLI: a command-line-interface tool that provides utilities, one of which is bootstrapping the development environment.
- Jest: a test runner developed by Facebook, which is widely adopted by modern developers.

5 Modelling and Implementation

This section discusses the concrete steps of developing the case study application with statecharts and Xstate. First of all, the process of translating the specification to the statecharts (modelling) will be discussed. Subsequently, this paper will provide details on the concrete implementation of the application, integration with the view layer, as well as an evaluation of the model. One typical enhancement of the application: matcher form, fetching album art and audio analysis from a third-party service provider, is included in order to observe how statecharts accommodates for specification change.

5.1 Specification

The application to be built is a local music player. The end goal of the application should be providing the user with a basic capability to play local music files in their computers, with basic control such as repeat, shuffle, play/pause, next, previous and skip to a specific time. The features can be stated in the form of user stories:

1. As a user, I can upload local music files.
2. As a user, I can select a music track and it will automatically play.
3. As a user, I can play and pause the current track.
4. As a user, I can skip to a specific time in a track.
5. As a user, I can choose to repeat a single track, or the whole playlist.
6. As a user, I can shuffle the playlist.
7. As a user, I can skip to next and previous track.

5.2 Modelling the application

With the requirements explicitly stated, the Statecharts for this application is ready to be built. Constructing a Statecharts typically includes four activities:

1. Identifying states
2. Identifying the set of events, conditions and transitions between those states
3. Identifying the actions associated with the states and the transitions.
4. Refining or clustering the states, then additional iteration.

5.2.1 Upload tracks

Let us start with the first requirement of uploading local music. Every load operation typically involves four states: **initial**, **uploading**, **success** and **error.** Since there is no requirement to display the failure or success status, a simplification to use only two high-level states: **initial** and **uploading** is preferred.

In the **initial** state, the system can respond to a **LOAD_TRACK** event, transition to **uploading** state and launch the action to upload the music track. In the **uploading** state, the system does not handle the **LOAD_TRACK** event, therefore disables the upload feature. The upload process eventually results in either a success or failure via the **LOAD_SUCCESS** or **LOAD_FAILURE** events, at which point the system transitions back to the **initial** state and becomes ready again for more upload. It makes sense uninterested of having the same song twice in our system, a guard (**isNotDuplicate**) is introduced that will prevents uploading a track once it is already uploaded.

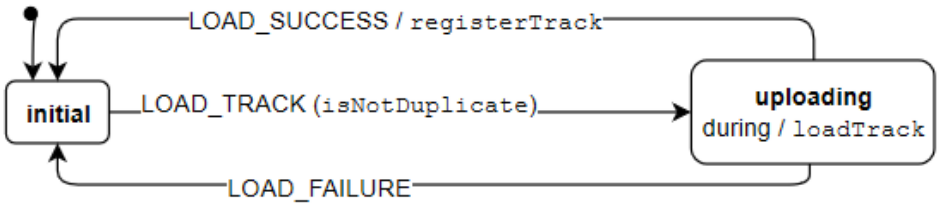

Figure 13. Statecharts for uploading music tracks

The actions identified for this requirement includes loading the track binary and updating the track listing. Two procedures **loadTrack** and **registerTrack** implement these behaviors. The former is an activity that keeps track of loading the song binary. It is provided a callback whose invocation requests the machine to send an event to its own, a technique known as broadcasting. The **LOAD_SUCCESS** and **LOAD_FAILURE** are

broadcasted to the machine when the underlining browser (in this case, the newly created audio element) emits the **canPlayThrough** event, or throwing an exception. The latter mutates the machine's context, a stateful data repository, to update the successfully added song

Once the capability to have music in the system is achieved, the core functionality of playback shall be the next goal. Regarding requirement 2, the ability to select a track depends on the existence of at least one track. Two states can be identified: **empty** and **ready**. In the former, all features are disabled. There is a transition to the **ready** state from the **empty** state caused by the first track upload being successful. Furthermore, since the user can always upload more tracks, the upload states are independent of the **empty** and **ready** states. These are clustered underneath **main**, which are orthogonal to the super-state **upload** containing the **initial** and **uploading** states. The resulting adjustment is illustrated in figure 14.

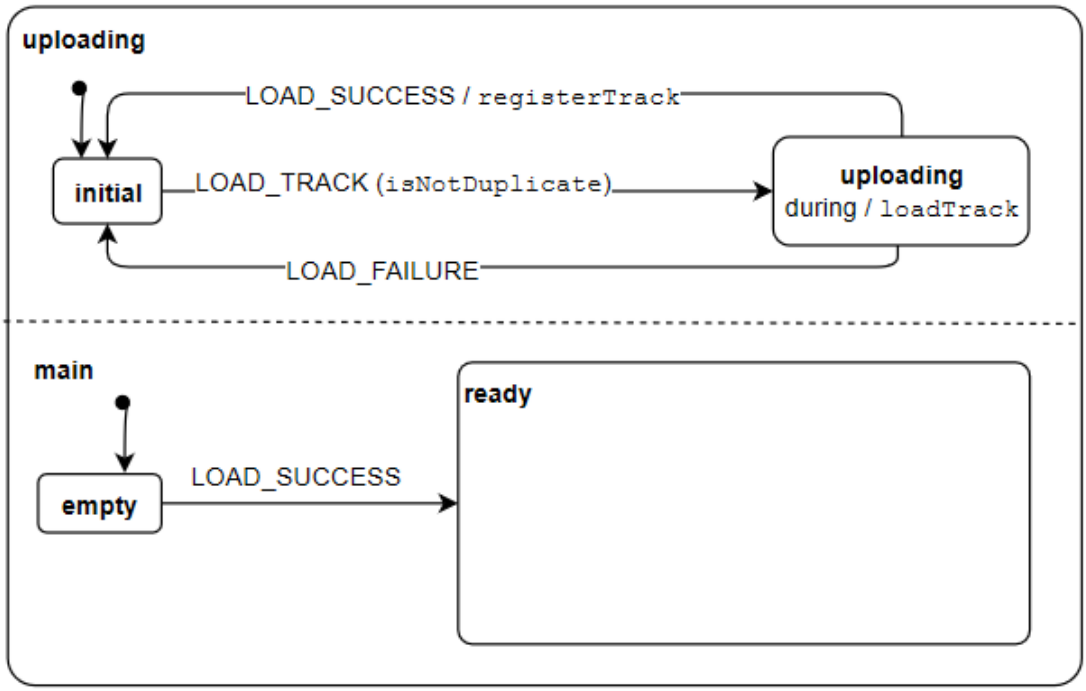

Figure 14. The application statecharts modelling uploading and main states.

5.2.2 Playback

Regarding play/pause feature, two major states can be identified: **playing** and **paused**. The initial state will be **paused**, because that is also the default state of the underlining browser media playback system upon the creation of a track. The related events can also be identified immediately as well: **PLAY**, **PAUSE** and **END**, which happens when the current track reaches the end.

The naïve approach thinks that the **playing** state is directly joined with the **paused** state by the **PLAY** event. This assumes the infallibility of the triggered effect. Should the action **play** fail, and unfortunately it might, this model would become inconsistent with reality. The state **attemptingPlay** is introduced as a remedy. In this state, an activity asks the system to play the audio binary and handles the failure, if any. This allows the model to sync the real playing status of the underlining system and therefore avoid introducing the aforementioned out-of-touch pitfall.

The keen readers might have a question regarding the action of pausing the track. According to the latest web standard, the call to pause a playing media in the browser is guaranteed to be successful [14], therefore, a direct transition with a fire-and-forget action is sufficient. Furthermore, **attemptingPlay** being a transient state should be unable to house any activity. In general, activities take non-zero amount of time to execute, but it is perfectly normal for an activity to be instantaneous. This is akin to a Javascript promise that instantly resolves. In fact, **attemptPlay** calls and waits for such a promise.

A user may select a track whenever they see fit as long as the track exists in the registry. The state may either respond to the event **SELECT_TRACK** when the application is **ready**, or in the **ready**'s many substates. When there are a hierarchy of states accepting an event, the most specific with the highest depth should be preferred. Thus, the trio **play/paused/attemptingPlay** can be clustered under a super-state, **playback** that accepts and handles the **SELECT_TRACK** event.

When the user switch track, the following are usually expected: the previous track is stopped (1) and the elapsed time reset to zero (2). Also, the specification calls for the selected track to be automatically played (3). These three actions happen in this specific order.

To satisfy such ordering, all three exit, transition and entry action types are needed. Upon exiting the **playback** state, the current track is paused. A parent state is exited after the children state exits, therefore the **resetToZero** action is appropriate as an exit action of the **playback** state. During the transition back to itself, the new track is selected. An entry event causes the newly selected track to be played automatically.

Pausing the track might either be an exit action placed on the **playing** state, or its parent, the **playback** state. Since all out-going transitions from the **playing** state, either to paused or via exiting the parent state requires pausing the track and the otherwise, placing the exit action on the **playing** state is the better choice.

The behavior 2 requires the same **resetToZero** action akin to when the playing track ends, on top of updating the current song to the selected song. These follows the exit action from the **playing** state, therefore are both appropriate being a transition actions associated with the **SELECT_TRACK** event.

To satisfy the behavior 3, the **playback**'s initial child state can be modified from **paused** to **attemptingPlay**. The transition will cause the state playback to exit and reenter itself, thus immediately enters **attemptingPlay** and cause the newly selected track to play. Figure 15 represents the resulting **playback** states.

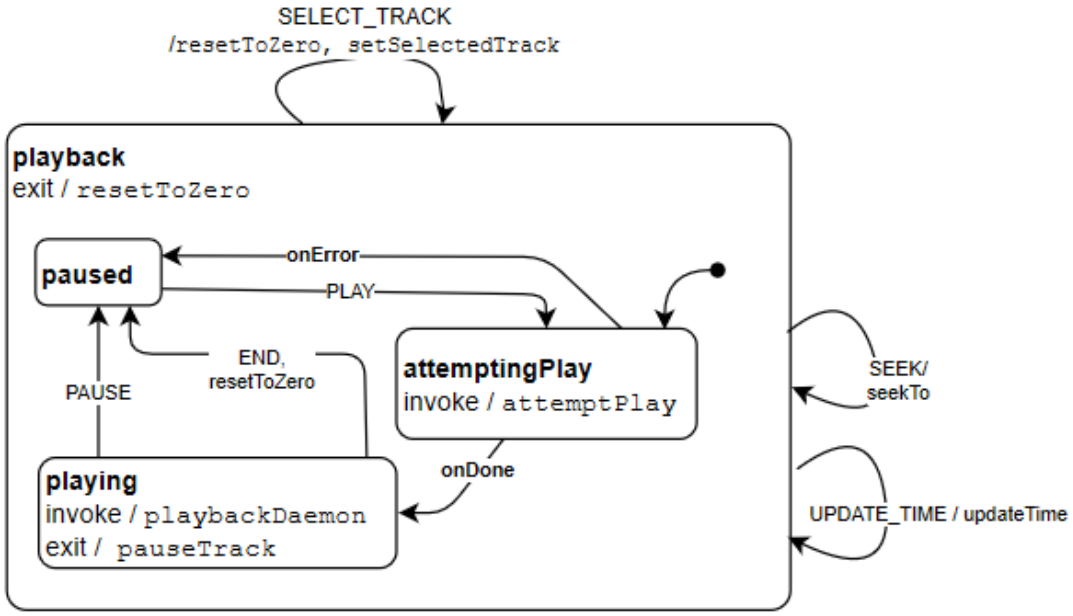

Figure 15. The statecharts for playback related state.

5.2.3 Monitoring elapsed time

Feature 3 calls for the ability to monitor the time elapsed in the current track and random seeking. It is reasonable to be able to seek irrespective of play or paused status, therefore the **SEEK** event is applicable to the super-state **playback,** similar to the **SELECT_TRACK** event**.**

Monitoring the elapsed time takes a non-zero amount of time. This task falls into the activity classification. Activities can be established by a pair of actions: the initiation executed at the entry of a state, and the cleaning-up at the exit of the state. This activity subscribes to the current active audio element of the selected track to receive update from the elapsed time via a native event**,** which in turn updates a value in the context. The adjustment is reflected in figure 15.

5.2.4 Repeat

Let us move forward to address the repeat and shuffle features. From the requirements, there are two types of repeat and a default disabled state. A **TOGGLE_REPEAT** event can be sent from the user, for example, via a repeat toggle button, to move between the three states one at a time. The next step is to identify the possible events for this particular set of states.

- **TOGGLE_REPEAT**: Toggle between the three repeat states: **repeatOnce**, **repeatAll** and **noRepeat** states
- **NEXT**: Wrap back to the beginning of the track or next track, depending on the repeating mode
- **PREV**: Similar to **NEXT**, but with previous track rather than the next track
- **END**: In effect the same behavior as the **NEXT** event.

Repeat all is a feature meant for a playlist whose size is greater than one. Simply put, repeat all is identical to repeat once if there is only one track in the playlist. Although, the effects that the **NEXT**, **PREV** and **END** events causes in the **repeatAll** state are unique: potential warping back to the beginning or the end of the playlist, meaning that consolidation of **repeatOnce** and **repeatAll** cannot be done. It is then necessary to introduce guards, used for detecting whether the current playlist length is greater than one, in the transition from **repeatOnce** to **repeatAll**.

The above figure also considers the effects that will be triggered in response to events in each state. The thought of the loop attribute of the underlining HTMLAudio element may arise and although achievable, several complications presents:

- Cleanup required: The loop attribute should only ever be set on the current active track and while the **repeatOnce** is the active state. If the state is transitioned away or another track gets selected (from feature 2), the attribute needs to be flipped to false before the next (or previous) track commence playing.
- Leaking responsibility: For the attribute to be cleaned up, either the **repeatOnce** state must also handle the **SELECT_TRACK** event, or **selectTrack** action must

also be overloaded with the aforementioned clean up task. Neither are particularly appealing, since bountiful unpleasant surprises and at the level of implementation requires attention during the model designing process, not to mention the tight coupling with the **playback** state.

- Single responsibility: Constructing a complex behavior from simple tasks are the gist of single responsibility pattern. Thus, it is more logical to construct this behavior from a set of simple, straight-forward actions rather than a new behavior, especially when clean-up is required.

In conclusion, the desired effect should be implemented via the composition of already available actions. Since the ability to play, pause, select a track, and skip to a location in a track (specified in req. 7) is already available, the behavior of repeating the current song can simply be resetting the current time to zero.

By the virtue of composing simple actions, the of associated actions for the **repeatAll** state can be built completely from reusable units. The act of selecting the next and previous track can be built on top of the random track selection already available. Figure 16 reflects the final result of the **repeat** state.

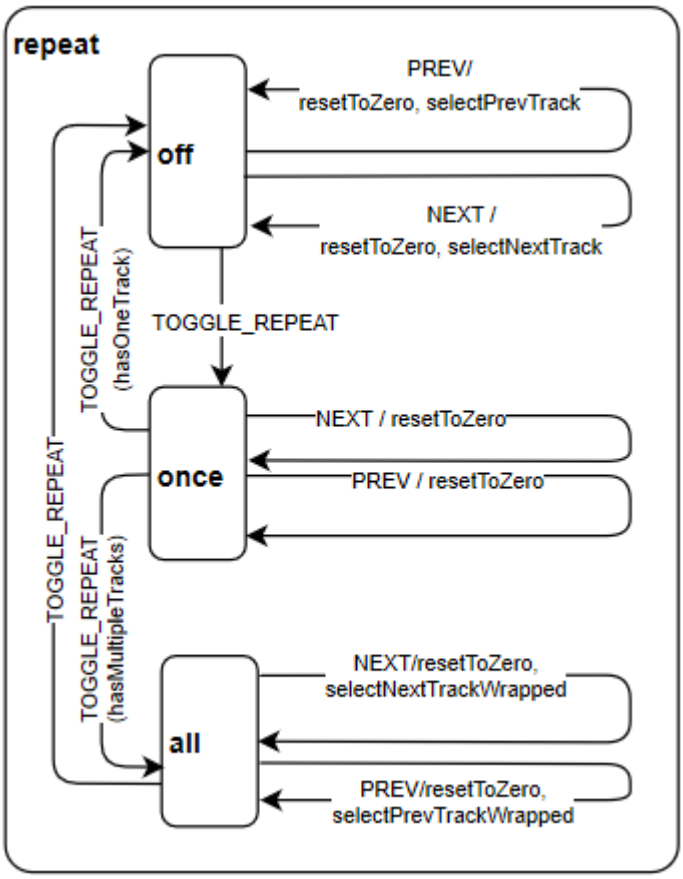

Figure 16. Statecharts concerning the repeat feature\

5.2.5 Shuffle

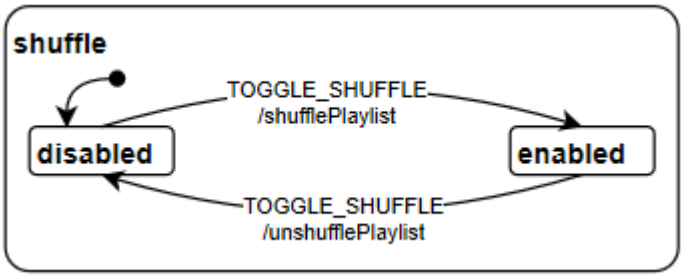

Figure 17. Statecharts concerning the shuffle feature

The design for the shuffle feature strikes great resemblance to that of repeat. For the sake of brevity, let us start with a finished diagram for this behavior. Shuffle is well known feature in music apps whose behavior is simple. However, one notable consideration should be considered during the design: what happens if there are additional songs

uploaded when shuffle is on. At this point, more clarification is needed from the requirements. This is desirable, because the modelling process has uncovered an ambiguity in the specification.

Let us refine requirement 5 such that when new songs are uploaded, their relative orders are preserved and appended to the shuffled list. Should the user desires another completely shuffled list, he or she can toggle off, and then on once more. This translates to no additional actions to be modelled.

The final task is to figure out the relationship between **playback**, **repeat** and **shuffle** all within the **ready** state. It can be clearly seen that a player can be in all the above states at once, therefore an orthogonality relationship is the right conclusion. Figure 18 is the comprehensive picture of the application behavior, satisfying the seven features established.

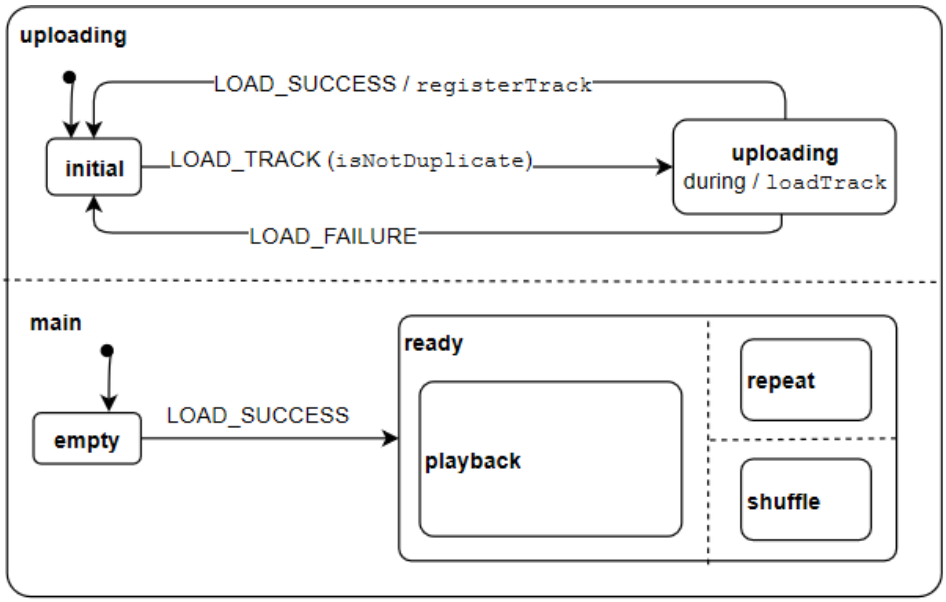

Figure 18. The application statecharts. The children states of playback, repeat and shuffle has been zoomed out for brevity.

5.3 Reviewing the model

The statecharts modelling the control of the music player application provides a structure that may be checked, debugged, or even formally verified. Due to the difficulty of formal

proving and the manageable complexity, the music player application can be subject through a debugging instead. The desired outcome of this task should be mostly the validation of the model. Four properties are of interest:

- Validity: The model accurately reflects reality.
- Reachability of states: all states should be reachable.
- Deadlock states: Identify states that may trap the application, especially undesired states.
- The model is deterministic.

The model is constructed on a number of assumptions. These assumptions can be captured by the list of requirements (1), as well as the ability of the application to provide recognizable events to the state machine (2). A reference to the web standard media playback documentation indeed confirms the browser capabilities with respect to assumptions (1). Recognizable events can be translated from native DOM events by the Vue components, which proves (2). These conclude that the model is credibly valid.

Next, let us evaluate the reachability of the states in the model. A state is reachable with respect to an origin state if there exists a path between the two states. For the purpose of this thesis, a formal proof shall not be attempted. What could be attempted instead is a manual enumeration of each individual traces from the origin point to the most significant states, such as **ready**, **playing**, **success** (in **unloadable**), etc. Although such method provides little guarantee and should not be employed for mission-critical system, the author finds that it is acceptable for this project. The author has found no faults in reachability of the model so far.

Another point to review is the existence of any deadlock state. Deadlock is not exclusive to a single state. A group of states can transition between each other, but not outside. Two useful indicators of deadlock states are the set of out-going arrows, and the conditions on the transitions. If the state is not final, and there is no out-going arrow, the analysis has detected a deadlock state. It is worth noting again that transition from parent

states counts as well. A transition might be blocked by a guard, and so it is worth paying close attention to guarded transitions.

Statecharts may be non-deterministic if the next state depends on extra information besides the guard, event, and the current state. Multiple out-going transition from a state to other exclusive states, where more than one associated guard might be true simultaneous is the second cause. For this model, the author detects no such breach in so far.

5.4 Effect implementation

The resulting model from earlier work produced a clear and well-defined list of effects the state machine can exert on the environment in order to deliver the required features. The knowledge of when and what the effects achieve has been made clear. What is left to be done is the concrete implementation of these effects. This section discusses in greater details how the system achieves its uploading and playback capability.

Some of the actions perform reading, writing and data manipulation. In Xstate, stateful data are stored in the context object. Storing data elsewhere is possible, however undesirable because it is more challenging to set up observability between these data with the view layer and a significant complication in serializing the whole state machine, should persistence be considered. In order to update these data, a pure function can be provided to the action constructor **assign** provided by Xstate. Xstate's **assign** function provides the current context, making it easier to test the action independently. Similarly, the **send** action creator provides the event broadcasting capability.

Most modern browsers support audio playback. There are two ways a blob of audio data might be played in the browser: using the HTMLAudio API or Web Audio API. Web Audio API allows sophisticated audio generation, manipulation, and playback. HTMLAudio API provides a simple method to control the playback of audios fetched from a resource origin [14]. For that reason, HTMLAudio is the technology of choice.

```
type Track = {
    id: string;
    audioElem: HTMLAudioElement;
    duration: number;
```

```
}
type Context = {
    tracks: Array<Track>,
    currentTrack: number, // the index of current track
    currentTime: number, // elapsed time of current track
    originalTracks: Array<Track> // a backup track array for shuffling
}
```

Listing 3. The type specification of the context object used by the application statecharts.

Table 1 lists the identified effects, the influence description, and their implementation.

Name	Influence	Implementation
`loadTrack`	Initiates and manages the underlining DOM mechanism for loading the audio files into memory	Initiates an HTMLAudioElement with the source from the input event.
`registerTrack`	Handles the loaded audio files, does whatever transformation necessary, and appends into the list of uploaded tracks ready for playing.	Appends into the variable `tracks` the payload of the event.
`setSelectedTrack`	Updates a variable tracking the currently selected track	Sets `currentTrack` to one retrieved by the index from the event payload and the `tracks`.
`attemptPlay`	Calls the underlining DOM Audio API to play the currently selected track	Calls `play` method of the `currentTrack's HTMLAudio` object. This method may throw.
`pauseTrack`	Calls the underlining DOM Audio API to pause the currently selected track	Calls `play` method of the `currentTrack's HTMLAudio` object. This method may throw.
`resetToZero`	Sets the current elapsed time back to the beginning	Sets `currentTime` and the similarly named attribute of the currentTrack's `HTMLAudio` object to zero.
`selectNextTrack`	Broadcasts `SELECT_TRACK` event with the event payload set to either the next in the playlist or wraps back to the beginning.	As described.
`selectPrevTrack`	Is similar to `selectNextTrack` but set to the previous or wraps to the end of the playlist	Calculates the previous track index, which is `currentTrackIndex` - 1 if not first and last index otherwise, then broadcast `SELECT_TRACK` with the result index.

`shuffle-Playlist`	Randomizes the order of tracks and keeps a reference to the old order for undo.	Generates a new track list with randomized order. The original track is saved in a temporary variable `originalTracks.``
`unshuffle-Playlist`	Restores the original order of the playlist.	Sets the variable tracks to that of the variable `originalTracks`.
`playback-Daemon`	Subscribes to a DOM event to periodically receives the current time and updates that to a variable. Broadcast `END` event if track is ended.	Broadcast `UPDATE_TIME` whenever the browser fires the `timeupdate` event. Fire `END` whenever the browser fire the `ended` event time is equal to the track duration
`seekTo`	Seeks to a specific time	Sets the `currentTime` of the `currentTrack`'s `HTMLAudio` objectto the specified time in the event payload.

Table 1. Actions and activities implementation.

Indeed, as covered in the theoretical background, a state machine may choose to emit events in response to the input event. The leveraging of that behavior to our advantages is seen with great clarity here. Of course, with the introduction of more events, for example **UPDATE_TIME**, comes additional adjustment of the model. This is a natural and arguably most modelling (and design) activity happens in such an iterative manner.

5.5 Integration with the view layer

Completing the model and the action implementation signifies the readiness of the control layer. Provided the right events, the application may finally start making pleasant noise. For that, the user interface needs to be developed and integrated with the executable Statecharts. The mechanism of integration, as well as an example usage in a Vue component are discussed in this section.

So far, only the model has been defined in a structure constructed with Xstate's **Machine**. It is necessary to instantiate a running instance of this **Machine**. Xstate provides the method **interpret**, which takes a **Machine** object and return a running instance. The method **start()** initiates the execution of the instance, termed service, and the method **onTransition** can be used to push the latest state of the service, as well as context data, upon every transition taking place. This will be the key integration point to the view layer.

As mentioned in chapter 5.2.2, Vue components may subscribe to changes in a Vuex store and update the controlled DOM with new data. Leveraging this mechanism, the machine state and context data can be made available to a Vuex store, and subsequently updated on every change by the **onTransition** method. Events emitted from the view layer are transformed into recognizable messages and fed to the machine, driving the application state forward. Defined here is a Vuex action that forwards these messages to the state machine. Upon state transitions, a mutation will be invoked by the state machine to save the latest state and stateful values into the store, which components can subscribe to on change. Listing 3 demonstrates an example integration implementation.

```
import Vuex from "vuex";
import Vue from "vue";
import { musicPlayingService } from "@/machine.js";

Vue.use(Vuex);

const store = new Vuex.Store({
  state: {
    currentState: musicPlayingService.initialState
  },
  actions: {
    sendEvent(ctx, event) {
      musicPlayingService.send(event);
    }
  },
  mutations: {
    updateState(state, payload) {
      state.currentState = payload;
    }
  }
});

musicPlayingService
  .onTransition(state => { // register new states to be up-
dated to Vuex store
    store.commit("updateState", state);
  })
  .start(); // start the FSM and

export default store;
```

Listing 4. Integration between Xstate and Vuex

Let us consider an example of a bar component where most of the control buttons of the player are located, namely the play/pause, next, previous, repeat and shuffle buttons. The entire bar should not be rendered, if there is no playback control. A conditional render using Vue's **v-if** directive can be put on the top-level div element. The condition would be the system being in the **ready** state. Here the advantage of the model can be seen in action, as the condition is succinct and understandable. The current state of the machine can be accessed by the **currentState** field. Since states are hierarchical and may include orthogonal parts, a convenient **matches()** method which accept a string can be used to produce the needed condition. The code section in listing 4 illustrates a small part of the implementations. Similar steps applied to other control buttons, such as repeat and shuffle.

```
<template>
  <div
    v-if="currentState.matches('main.ready')"
    class="buttonBar"
  >
  <prev-button @click="goPrev">
  <div>
    <play-button
        v-if="currentState.matches('main.ready.playback.paused')"
        @click="requestPlay"
    />
    <pause-button
        v-if="currentState.matches('main.ready.playback.playing')"
        @click="pause"
    />
  </div>
  <next-button @click="goNext">
</template>
<script lang="js">
import { PlayButton, PauseButton, NextButton, PrevButton } from '@/com-
ponents/ControlBarButtons'
export default {
    components: {
        'play-button': PlayButton,
        'pause-button': PauseButton,
        'next-button': NextButton,
        'prev-button': PrevButton
    },
    methods: {
        requestPlay() {
            this.$store.send('PLAY');
```

```
        },
        pause() {
            this.$store.send('PAUSE');
        },
        goPrev() {
            this.$store.send('PREV');
        },
        goNext() {
            this.$store.send('NEXT');
        }
    },
    computed: {
        currentState() {
            return this.$store.currentState
        }
    }
}
</script>
```

Listing 5. Example implementation of a control bar in Vue driven by Xstate

The clarity gained from the introduction of Xstate and the statecharts model developed earlier cannot be understated. The component depends on a much smaller amount of conditions for the rendering, and the event handlers are straight-forward. The relationship between the states are simplified to the view layer so much so that the play and pause buttons do not require the knowledge that the play and pause are two mutually exclusive states. All the effects are handled by the control layer, and no complex business logic is distributed into the view layer.

5.6 Extending the application

5.6.1 Specification

One particularly attractive claim from statecharts is its enablement to scaling and maintenance. Let us put that claim to the test by examining a typical enhancement to this project. A typical music player application as of today is expected to provide additional musical knowledge. In order words, the user expects to see the album cover art, the artist name, and possibly other tracks in the album. Let us extend the specification with two additional requirements:

8. As a user, I can see the album art for the currently selected track, as well as the artist name.

9. As a user, I can see the audio analysis and other tracks in the album of the currently selected song.

Requirement 8 calls for the extraction of audio file tag metadata in order to obtain the artist name as well as album art, if available. Feature 9 requires integration with a musical knowledge database. These requirements present a typical problem faced during the development of a web application: managing network calls. This forms an adequate yet pragmatic challenge to the usability of statecharts and Xstate.

5.6.2 Modelling

Commencing the work is the task of evaluating the general strategy of how the requirements could be fulfilled. For feature 8, there exists a Javascript library that enables extracting audio file metadata. The extraction does not guarantee the result, because some track may have any embedded metadata. Regarding feature 9, a musical service provider shall be employed. In this project, Spotify will be the service of choice.

For the sake of simplicity, the process for a singular track will be first considered. The general idea is that the extraction of metadata be performed as part of the track loading. The application attempts to produce a best guest to match with the external service's entries and presents the user with the suggestion. If that fails, the user can manually match and retrieve the corresponding musical knowledge from Spotify. After the confirmation, the app attempts to retrieve additional data about the track from the third party service provider. Figure 19 illustrates the general process.

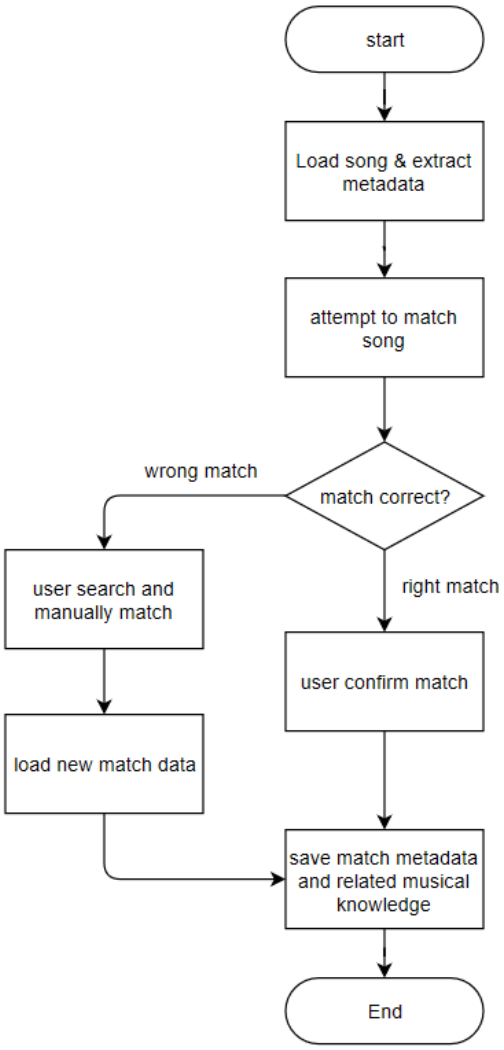

Figure 19. High-level flow chart of requirement 8 & 9. Certain processes are presumed to be unfillable for brevity.

However, our application should be capable of handing the enrichment process of multiple track at once. This is a reasonable thought, because the process of loading, extracting metadata, and enrichment of each track only has the track and the server API as the external dependencies, all of which are not shared with any other track. In order words, the process can be achieved concurrently. Since the number of concurrent processes are non-determined, orthogonality is not the correct solution. As introduced in chapter 3.6, this problem is best approached using communicating actor state machines(2).

Needless to say, the upload state no longer captures the reality of how the application behaves. Thus, several adjustments to the statecharts are required. Within the main

machine, a state can be defined in which the machine is capable of responding to the event **UPLOAD_TRACK**. The machine handles this event by creating and distributes task to a new child machine. The child machine implements the process of loading and enriching the track. Frequently, the child machine sends events back to the main machine at points of interest, such as when the metadata has been extracted successfully, etc. The success and failure of the entire loading and enriching process are captured in the states of the child machine and are moved out of the main statecharts. At this point, the application is able to handle the **UPLOAD_TRACK**, as well as events sent from child machine. This means that the uploading states may be simplified to a single state, **uploadable**. Figure 20 and 21 reflects the adjustment for the application state machine.

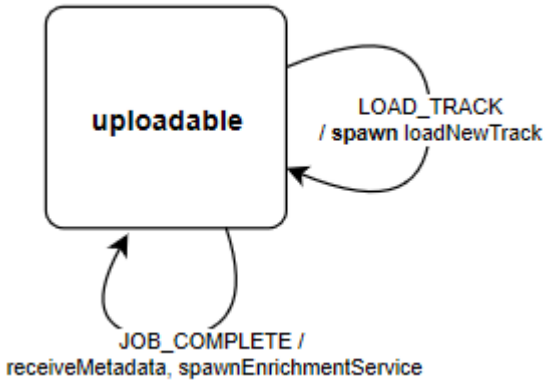

Figure 20. Revised `uploadable` state to chiefly handles supervision and distribution of computation to children state machines

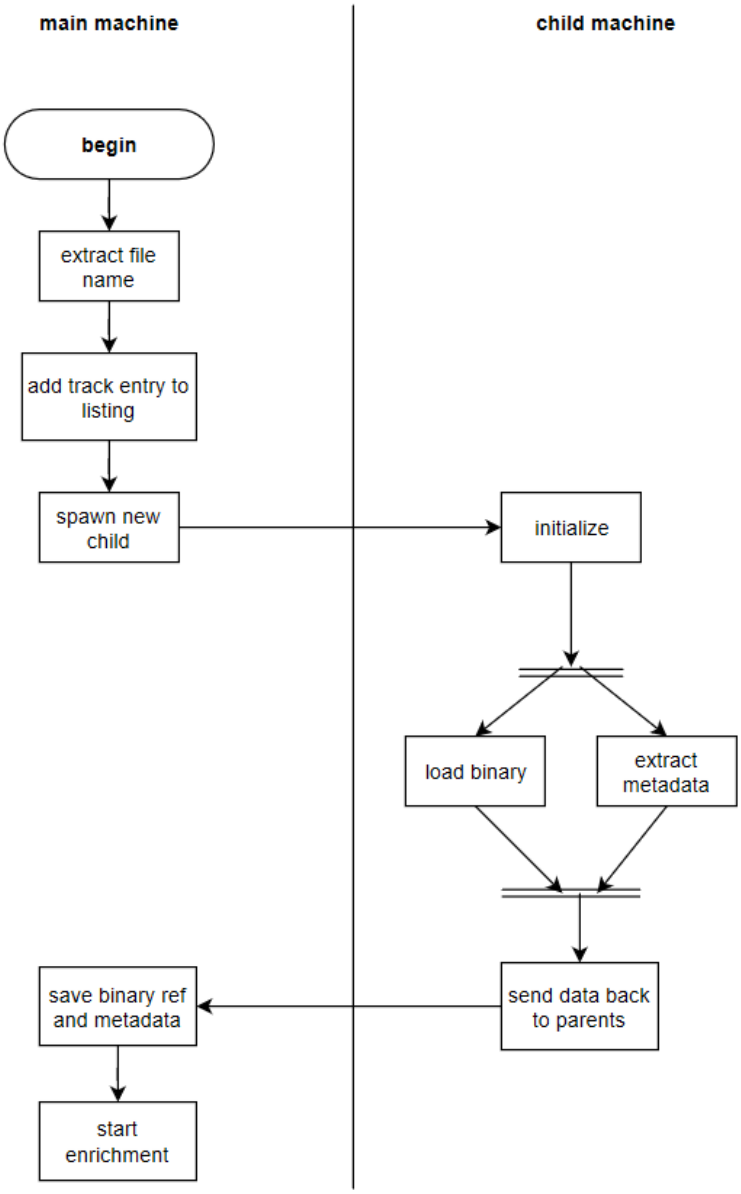

Figure 21. The flow chart of the process in loading and extracting metadata of new tracks

In additions to the adjustment made to the uploading processes, the playback aspect of the system requires similar refactoring. Due to the fact that the track is made visible in the playlist even before the binary is loaded, the playback state should incorporate the new loading state. When the user selects a new song, the machine checks if the audio binary is loaded from the underlining DOM, and transitions accordingly. Figure 22 reflects the adjustment to the **playback** state.

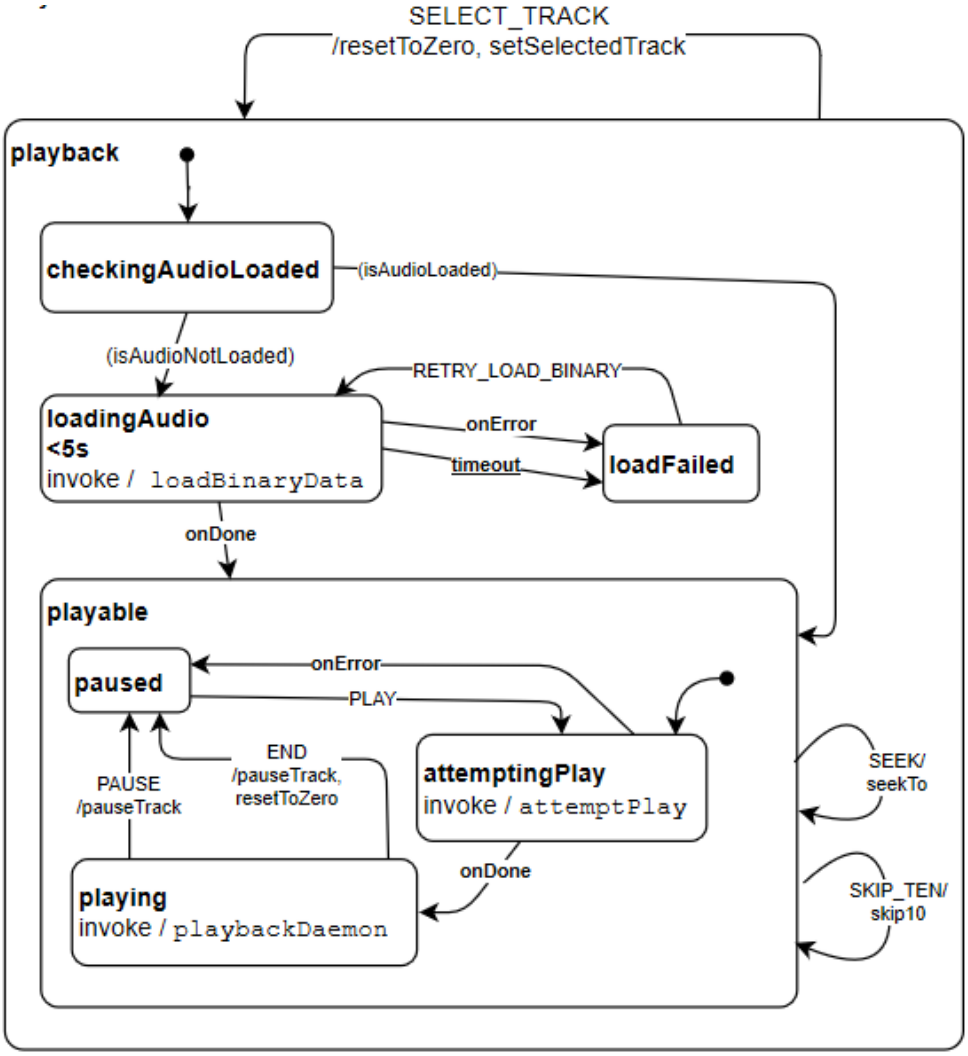

Figure 22. The adjusted `playback` state, taking into account the state of song binary loading or failure of such loading.

Requirement 9 introduces a situation where computations that happens in the child machine is influenced by events, such as the request to change the matched song name, or confirmation of matches. As a result, the responsible child machine handling the enrichment processes is modelled explicitly as a state machine. Some of the events are not handled by the main machine and are forwarded to the corresponding child machine instead. Figure 23 shows the interfaced **enrichable** state and figure 24 is the statecharts of the child machine that handles per-track enrichment.

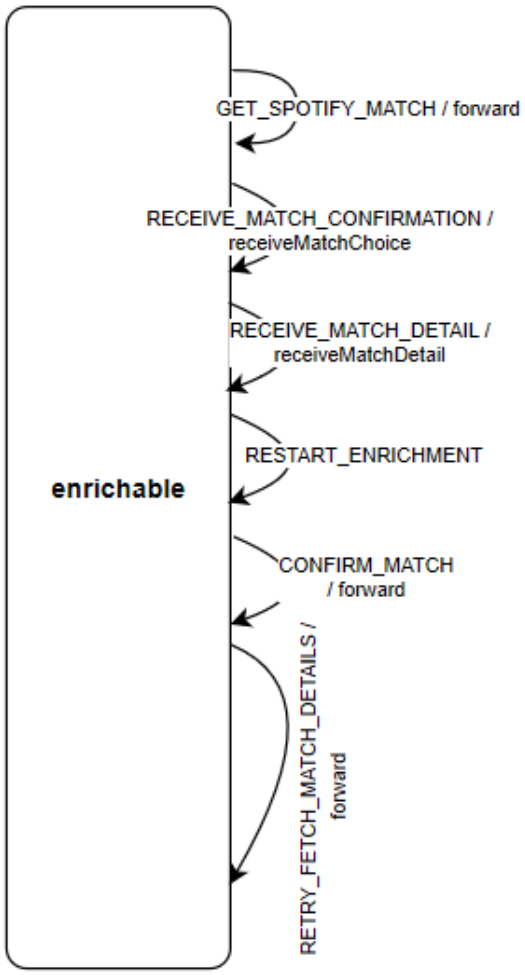

Figure 23. `Enrichable` state that interface with children machines

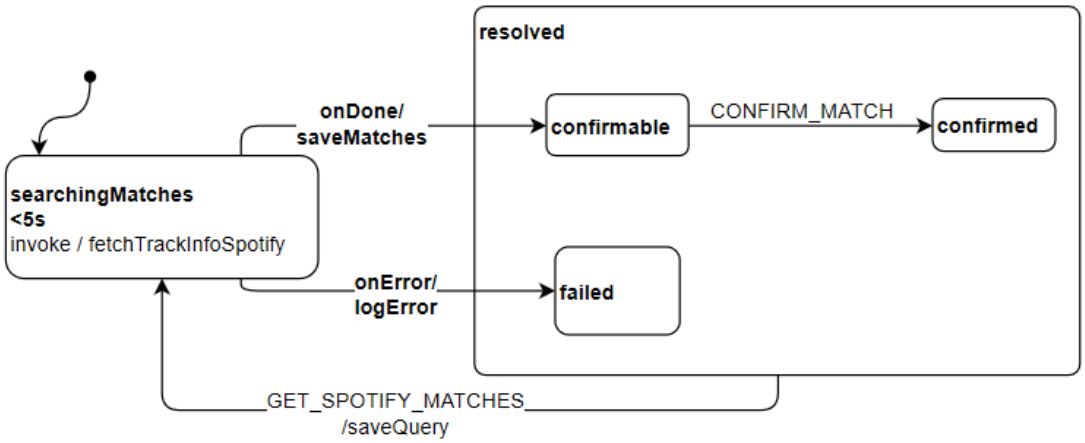

Figure 24. Enrichment child machine

5.6.3 Implementation

In Xstate, the children actor state machines may be implemented using several constructs depending on the nature of the communication. Promise and Observables are constructs that typically responds only once and does not accept events from the main (parent) state machine. Callback and Xstate Machine may both send and receive events to and from the parent machine. The child machine can be implemented as a callback that sends event at important milestone, such as a successful loading of a track or metadata, or corresponding failures. The child machine may be started using the **spawn()** action creator.

While it is possible to model children state machines with statecharts, recent development in Javascript, especially the introduction of Promise and async/await remarkably simplified the binding between the events, states, and actions involved in coordinating the asynchronous communication, resulting in a seemingly synchronous code performing retrieval or mutation of data. Therefore, the author argues that modelling the system with statecharts might not provide the most convenience, despite the event/state nature. However, should these children state machines receive events from the parents, and the states evolved well beyond the convenience offered by the language features, explicitly modelling with statecharts should be considered. The children machines for feature 8 and 9 are implemented as callback and Xstate machines, respectively.

The extension process concludes with the review of the model. This process is similar to what has been covered in chapter 5.3.3. For the sake of brevity, the author found no unreachable or unescapable states in both the main application statecharts and the children machine statecharts.

This extension case study demonstrated the advance usage of statecharts and Xstate in tackling a very typical concurrency scenario in web developments. The application can be extended to include scenario where sophisticated coordination is required. It can be seen that the enhancement process was with ease, while clarity on application behavior is preserved. In fact, a significant refactoring was achieved as well.

6 Result and evaluation

The implementation of the music player application provided invaluable insights into the strength of Statecharts for the development and maintenance web applications. With the conclusion of section five, a fully functional and adequately complex application with acceptable level of utility has been achieved, as seen in Figure 25 to 27. The entire main statecharts is shown in figure 28. The application can be accessed publicly at http://xtune.azurewebsites.net/.

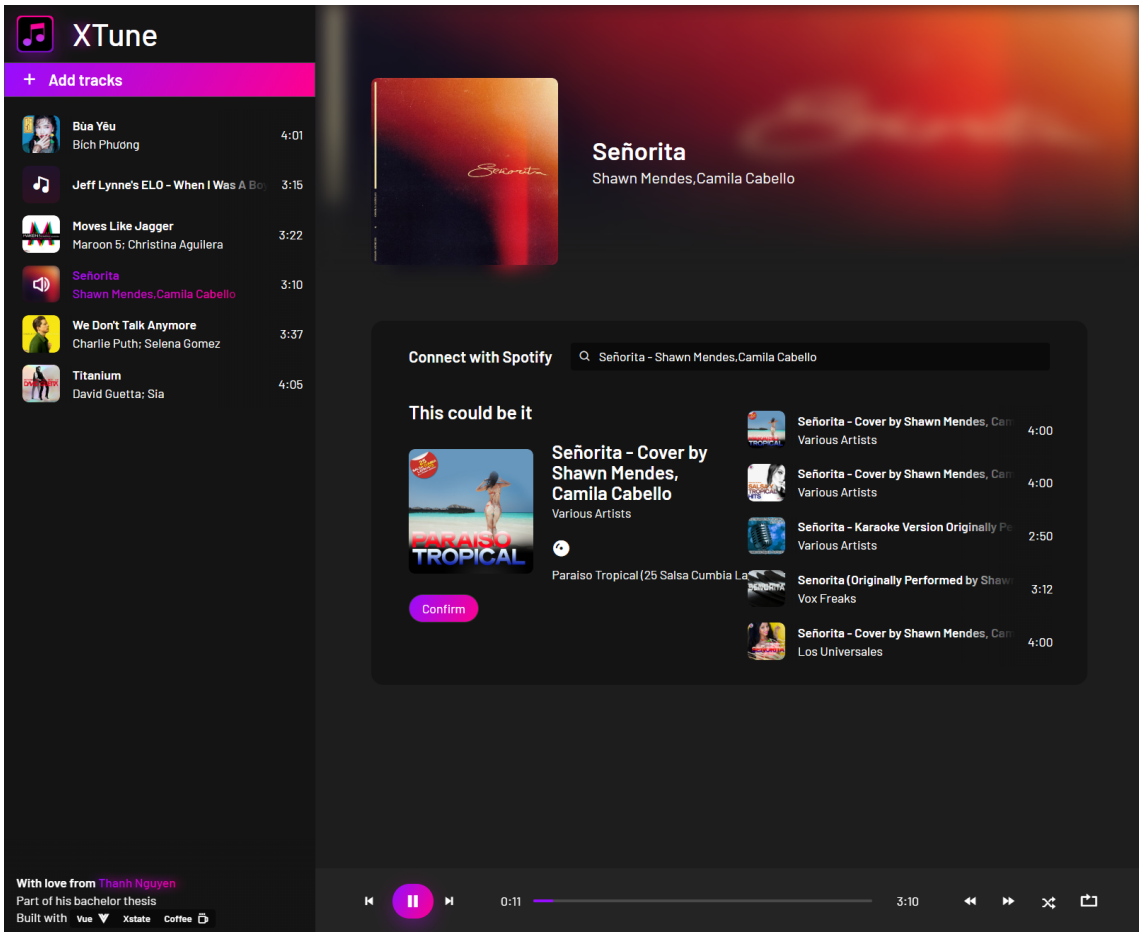

Figure 25. The case study music player application, when a track is playing and in the process of enrichment

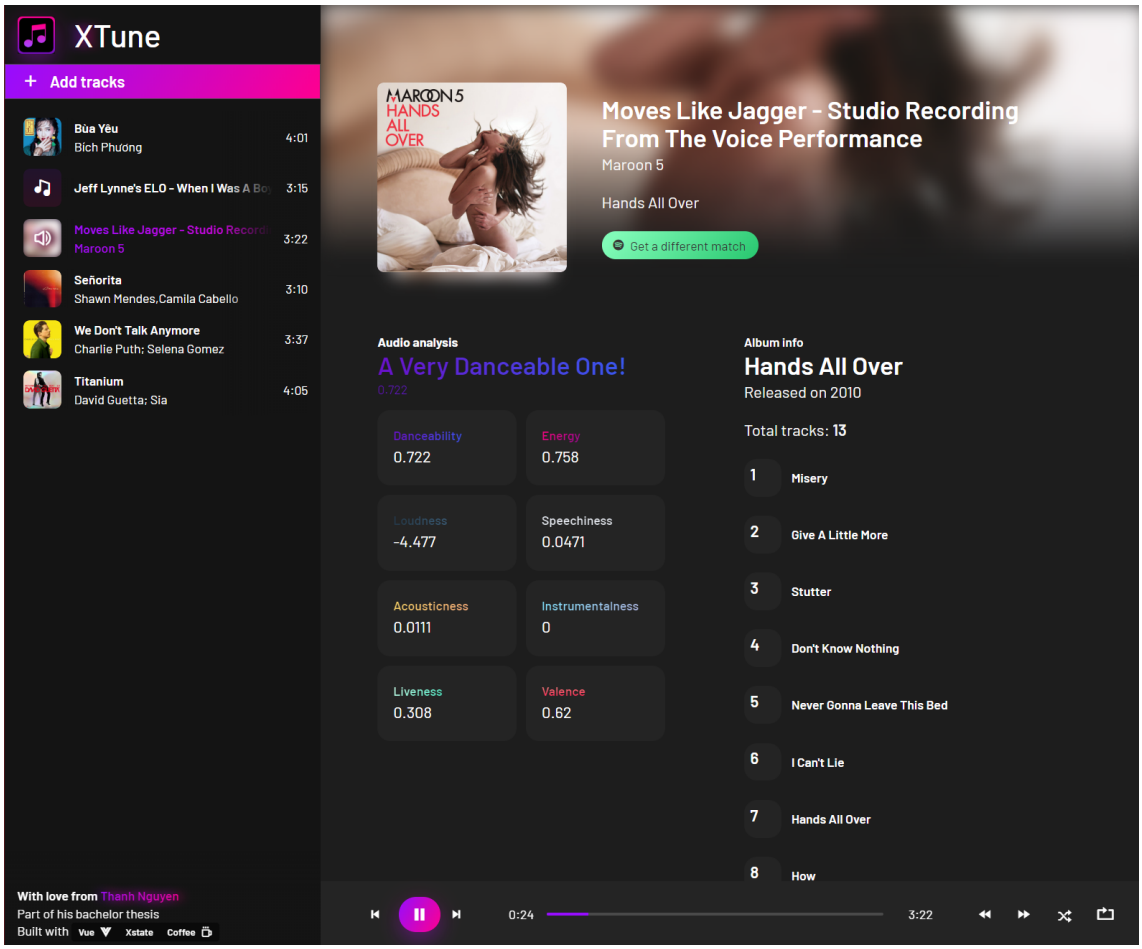

Figure 26. Playing an enriched track

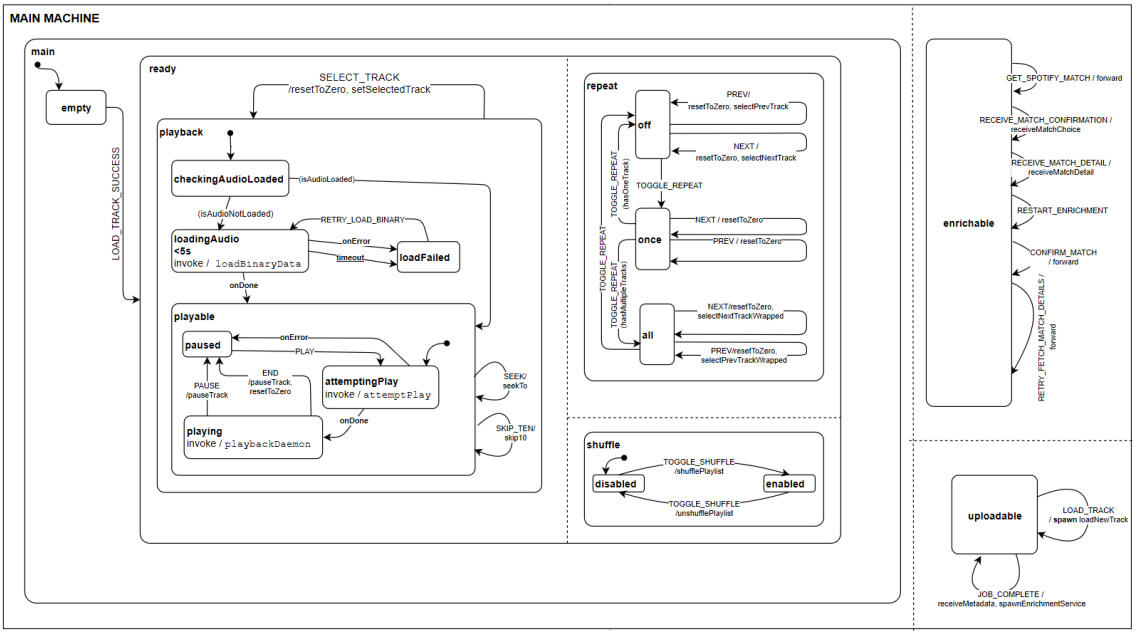

Figure 27. The main application statecharts, not including the diagram for children machines that handle loading or enrichment.

The development process of the application proved manageable. All the nine requirements are realized. Some of those, namely the last two, are non-trivial concurrency problem in fetching resources from the internet. Even so, their implementations even elaborated in sufficient details despite the length constrain of this thesis.

In the music playing application, it is evident that the introduction of an executable statecharts presents little changes in how the view layer is structured. In fact, the fundamental nature of the statecharts made no assumption about the view layer. This means that should the application be implemented using a different, or a combination of view frameworks, the statecharts still requires virtually no migration. Furthermore, applications built with statecharts are easier to migrate to a different platform. Baring the adjustment in adapting to how certain processes of the underlining platforms operates, most of the executable model can simply be ported as-is.

Beside the software code, one important artifact, namely the statecharts, is produced. The statecharts of the application serves as a formal, rigorous documentation of the application behavior. In this particular case when combined with the usage of Xstate, the statecharts is rendered both evergreen and interactive. Automated, detailed and evergreen documentation is invaluable in software development. This only serves as

additional evidence for the strengths of Statecharts in expressing the general behavior of the application, without code inspection.

Statecharts are remarkably useful in modelling reactive application, web included. However, the modelling competence and theoretical understanding requirement presents a considerable challenge to wide-spread adoption in the industry. There exists a threshold of complexity only above which Statecharts, and modelling in general, become useful. Simple application employing tools already incorporated state machines, such as async/await flow control syntax, may find little justification in investing in Statecharts.

All in all, the case study has been specified and implemented comprehensively. The author finds that Statecharts and Xstate provided critical benefits to the development of the case study, namely a more structured control with clear and precise relationship between state, data, events, and effects; evergreen documentation, and potential formal verification. It is safe to say that Statecharts (and Xstate) are a good fit for web development, with many attractive merits and therefore should be adopted if applicable.

7 Conclusion

In summary, the thesis introduced and elaborated on how a classic formalism, widely embraced in the most demanding software, such as in telecommunication infrastructure, aircraft, and spaceship, can be applied to building user interface on the web. In doing so, the science of finite state machine and state transition diagram, upon which Statecharts is built, was explained. The features of Statecharts, which is chiefly the addition of hierarchy and independence to state transition diagram and the problem that it solves, namely state explosion, were also discussed in detail. The thesis concluded with the concrete application of Statecharts a Javascript-based executable Statecharts engine, Xstate in developing a web music player.

Statecharts provides an expressive and powerful way to capture and model a reactive system at any scale without suffering from state explosions. The benefit of the underlining state transition diagram is preserved. This provides a clear and explicit model of how the application behaves, which assists in identifying potential logical defects and render the application conducive to modification, enhancement, and migrations. The

constructed model provides a target upon which formal verification maybe attempted so that certain critical features can be guaranteed, or at least model-driven automatic test generation. Statecharts may be daunting to introduce and utilize due to the prerequisite competence in modelling and related theoretical knowledge of finite state machine, however the author believes it is a necessary investment that will only benefit in the long term as the application grows in scale and complexity.

Realizing the usefulness of Statecharts to web application development, an increasing number of organizations is adopting the approach. Although an explicit guide is not given, introducing Statecharts into existing software is possible and affordable to do so. Section 5 can be used as a crude guide on how this task may be performed, and the Xstate community also provides helper tools to integrate with other prevalent state management solutions, such as Redux and Vuex.

Although the interest toward Statecharts is rising in the web application development community, the full potential of the principle has yet to be realized. More researches are needed in several fronts. One such may be the development of an open source model verification software tailored to web applications. Since Statecharts can be employed to analyze user flow, analytics tooling can be built for even more intuitive design.

References

1 J. Harris & A Patterson-Hine, “State Machine Modeling of the Space Launch System Solid Rocket Boosters,” NASA Aeronautics Scholarship Program – Internship Final Report, 2013.

2 Robert J. Hall, “Specification, validation, and synthesis of email agent controllers: A case study in function rich reactive system design”, 2000.

3 Christopher Dragert, J¨org Kienzle, Clark Verbrugg, “Statechart-based AI in Practice”, McGill University, 2012

4 Michael Westrate, “Pure rendering in the light of time and state” [Internet] [cited 30th December 2019]. Available at:

https://medium.com/@mweststrate/pure-rendering-in-the-light-of-time-and-state-4b537d8d40b1

5 “Redux – Motivation” [Internet] [cited 30th December 2019]. Available at:

https://redux.js.org/introduction/motivation

6 C.S Hendricksen, “Augmented state-transition diagrams for reactive software” ACM SIGSOFT Software Engineering Notes, 1989

7 Ian Horrocks, “Constructing the user interface with Statecharts”, Addison Wesley, 1999.

8 David L. Parnas, “On the use of transition diagrams in the design of a user interface for an interactive computer system”, ACM ’69: Proceedings of the 1969 24th national conference [p. 379-385], 1969.

9 E.M Clarke, E.A. Emerson and A.P Sistla, “Automatic verification of finite state concurrent systems using temporal logic specifications”, ACM Trans. Prog. Lang. Syst. 8 [p. 244-263], 1986.

10 J. Martin and C. McClure, “Diagramming Techniques for Analysts and Programmers” Prentice-Hall, 1985.

11 David Harel, “Statecharts: a visual formalism for complex systems*”, Prentice-Hall, 1985.

12 Raymond Roestenburg, Rob Bakker, Rob Williams, “Akka in action”, Manning, 2016 [p. 233 – 234].

13 “Xstate - Transition” [Internet] [cited 30th December 2019]. Available at:

https://xstate.js.org/docs/guides/transitions.html#wildcard-descriptors

14 “MDN Web Docs - Web Audio API” [Internet] [cited 30th December 2019]. Available at:

https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API