

DELHI TECHNOLOGICAL UNIVERSITY

# Finite State Automata Based Reactive Interface Design

---

Theory of Computation (MC-304)

**Anish Sachdeva**

DTU/MC/2K16/013





# Finite State Automata Based Reactive Interface Design

Theory of Computation (MC-304) Project

29<sup>th</sup> April 2020

---

Anish Sachdeva

DTU/2K16/MC/13

Delhi Technological University



# Acknowledgements

I would like to express my special thanks of gratitude to my teacher Prof Dr. Sangita kansal of the Mathematics Department who gave me a golden opportunity to do this wonderful project on Theory of Automata and its practical application .

In this project I had the opportunity to explore how we can use finite state automata and regular machines practical applications and also how the workflow of developers, especially user design developers who work closely with software developers to give their customers the best experience work can benefit from new and easy workflow design.

In this project I explored many popular open source libraries such as cyclejs - a popular framework library that implements graph based data structures. I also got my hands dirty with state-transducer which is a state representation library that can be used to define several states and symbols or *actions* that might lead to the defined *states*.

Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.



# Index

<b>Motivation</b>	<b>1</b>
<b>Introduction</b>	<b>2</b>
<b>Finite State Transducers</b>	<b>3</b>
<b>General Specifications</b>	<b>5</b>
<b>Implementation</b>	<b>7</b>
<b>Tests</b>	<b>8</b>
Test Strategy	8
Test Selection	12
Test Implementation	13
Integration Tests	13
<b>Modelizing the User Flow With Extended State Machine</b>	<b>14</b>
<b>Running the Project on Your Local Machine</b>	<b>16</b>
<b>Examples</b>	<b>17</b>
About Section	17
Questions Section	17
Teams Section	18
Review Section	19
<b>Conclusion</b>	<b>21</b>
<b>Bibliography</b>	<b>22</b>

## Motivation

There are many user facing frameworks out there in the world that each have their strengths and weaknesses and bring something to the table to ease client facing design and development and also increase the rate at which developers can prototype.

Some examples of these client facing frameworks are Angular, React, Vue.js, Handlebars etc. After using a few of these frameworks I have found that developing a flow as to where the user will be navigated to and based on the information present takes time implementing in any of the given above frameworks as these complex flow charts are eventually created using a multitude of switch and if blocks that make the code convoluted and harder to comprehend.

In this report using a combination of the open source cycle-js framework, the open source state-transducer library a new method of defining user experience and user flow is introduced in this project.

The new method uses simple json (Javascript Object Notation) objects to define the user states and the user flow given input/action and then this flow is implemented automatically on the browser without defining explicit routes.

Further, a mechanism to test the user flow that the developer has created is also provided within this project and this project lays down the foundation of the new proposed mechanism complete with end-to-end tests, automatic routing, integration tests and also unit tests.

This proposed mechanism and hypothetically ease development and also greatly improve the speed of development and prototyping.

## Introduction

This demo aims at showing how state machines can be used to modelize reactive systems, in particular user interfaces. They have long been used for embedded systems, in particular for safety-critical software.

We will use a real case of a multi-step workflow (the visual interface however has been changed, but the logic is the same). A user is applying to a volunteering opportunity, and to do so navigate through a 5-step process, with a specific screen dedicated to each step. When moving from one step to another, the data entered by the user is validated then saved asynchronously.

That multi-step workflow will be implemented in two iterations :

1. In the first iteration, we will do optimistic saves, i.e. we will not wait or check for a confirmation message and directly move to the next step. We will also fetch data remotely and assume that fetch will always be successful (call that optimistic fetch). This will help us showcase the definition and behaviour of an extended state machine.
2. In the second iteration, we will implement retries with exponential back-off for the initial data-fetching. We will also implement pessimistic save for the most 'expensive' step in the workflow. This will in turn serve to showcase an hierarchical extended state machine.

With those two examples, we will be able to conclude by recapitulating the advantages and trade-off associated with using state machines for specifying and implementing user interfaces.

The implementation uses cyclejs as framework, and state-transducer as a state machine library.

## Finite State Transducers

A finite state transducer essentially is a finite state automaton that works on two (or more) tapes. The most common way to think about transducers is as a kind of "translating machine". They read from one of the tapes and write onto the other. This, for instance, is a transducer that translates as into bs:



a:b at the arc means that in this transition the transducer reads a from the first tape and writes b onto the second.

Transducers can, however, be used in other modes than the translation mode as well: in the generation mode transducers write on both tapes and in the recognition mode they read from both tapes. Furthermore, the direction of translation can be turned around: i.e. a:b can not only be read as "read a from the first tape and write b onto the second tape", but also as "read b from the second tape and write a onto the first tape".

So, the above transducer behaves as follows in the different modes.

- generation mode: It writes a string of as on one tape and a string bs on the other tape. Both strings have the same length.
- recognition mode: It accepts when the word on the first tape consists of exactly as many as the word on the second tape consists of bs.
- translation mode (left to right): It reads as from the first tape and writes an b for every a that it reads onto the second tape.
- translation mode (right to left): It reads bs from the second tape and writes an a for every f that it reads onto the first tape.

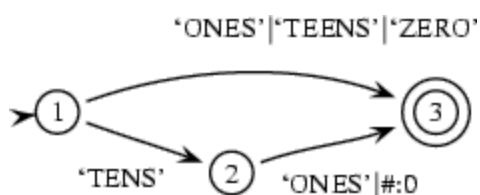
Transitions in transducers can make jumps going from one state to another without doing anything on either one or on both of the tapes. So, transitions of the form a:# or #:a or #:# are possible. Here is an example:



And what does this transducer do?

- generation mode: It writes twice as many as onto the second tape as onto the first one.
- recognition mode: It accepts when the second tape has twice as many as as the first one.
- translation mode (left to right): It reads as from the first tape and writes twice as many onto the second tape.
- translation mode (right to left): It reads as from the second tape and writes half as many onto the first one.

Similar to FSAs, we can also use categories to label the arcs and provide a kind of lexicon which translates these categories into real labels, i.e. labels of the form X:Y. Here is an example translating English number terms into numbers.



And here is the lexicon that maps the category labels to standard FST transition labels:

```

lex(one:1, `ONES') .
lex(two:2, `ONES') .
lex(three:3, `ONES') .
lex(four:4, `ONES') .
lex(five:5, `ONES') .
lex(six:6, `ONES') .
lex(seven:7, `ONES') .
lex(eight:8, `ONES') .
lex(nine:9, `ONES') .
lex(eleven:11, `TEENS') .
lex(twelve:12, `TEENS') .
...
lex(twenty:20, `TENS') .
...
lex(zero:0, `ZERO') .

```



Here are the initial specifications for the volunteer application workflow, as extracted from the UX designers. Those initial specifications are light in details, and are simple lo-fi wireframes.

[illegible]



In addition, the following must hold :

- it should be possible for the user to interrupt at any time its application and continue it later from where it stopped
- user-generated data must be validated
- after entering all necessary data for his application, the user can review them and decide to modify some of them, by returning to the appropriate screen (cf. pencil icons in the wireframe)

## Implementation

We use the stream-oriented `cycle js` framework to showcase our state machine library. To that purpose, we use the `makeStreamingStateMachine` from our library to match a stream of actions to a stream of events. We then wire that stream of actions with `cyclejs` sinks. In this iteration, we make use of two drivers : the DOM driver for updating the screen, and a domain driver for fetching data.

Code available in [dedicated branch](#).

## Tests

### Test Strategy

It is important to understand that the defined state machine acts as a precise specification for the reactive system under development. The model is precise enough to double as implementation for that reactive system (partial implementation, as our model does not modelize actual actions, nor the interfaced systems, e.g. HTTP requests, the network, etc.), but is primarily a specification of the system under study. In the context of this illustrative example, we used our state transducer library to actually implement the specified state machine.

It ensues two consequences for our tests :

- the effectful part of the reactive system must be tested separately, for instance during end-to-end or acceptance tests
- assuming that our library is correct (!), testing the implementation is testing the model, as the correctness of any one means the correctness of the other.

We thus need to test the implementation to discover possible mistakes in our model. The only way to do this is manually : we cannot use the outputs produced by the model as oracle, as they are precisely what is being tested against. Hence test generation and execution can be automated, but test validation remains manual.

That is the first point. The second point is that the test space for our implementation consists of any sequence of events admitted by the machine (assuming that events not accepted by the machine have the same effect that if they did not exist in the first place : the machine ignores them). That sequence is essentially infinite, so any testing of such a reactive system necessarily involves only a finite subset of the test space. How to pick that subset in a way to generate a minimum confidence level is the crux of the matter and conditions the testing strategy to adopt.

Because our model is both specification and implementation target, testing our model involves testing the different paths in the model<sup>1</sup>. Creating the abstract test suite is an easily automatable process of simply traversing through the states and transitions in the model, until the wanted model coverage is met. The abstract test suite can be reified into executable concrete test suites, and actual outputs (from the model implementation) are

compared manually to expected outputs (derived from the informal requirements which originated the model).<sup>1</sup>

- All states coverage is achieved when the test reaches every state in the model at least once. This is usually not a sufficient level of coverage, because behavior faults are only accidentally found. If there is a bug in a transition between a specific state pair, it can be missed even if all states coverage is reached.
- All transitions coverage is achieved when the test executes every transition in the model at least once. This automatically entails also all states coverage. Reaching all transitions coverage doesn't require that any specific sequence is executed, as long as all transitions are executed once. A bug that is revealed only when a specific sequence of transitions is executed, is missed even in this coverage level. The coverage can be increased by requiring :
  - All n-transition coverage, meaning that all possible transition sequences of n or more transitions are included in the test suite.
  - All path coverage is achieved when all possible branches of the underlying model graph are taken (exhaustive test of the control structure). This corresponds to the previous coverage criteria for a high enough n
  - All one-loop path, and All loop-free paths are more restrictive criteria focusing on loops in the model.

Miscellaneous model coverage criteria<sup>2</sup> are commonly used.

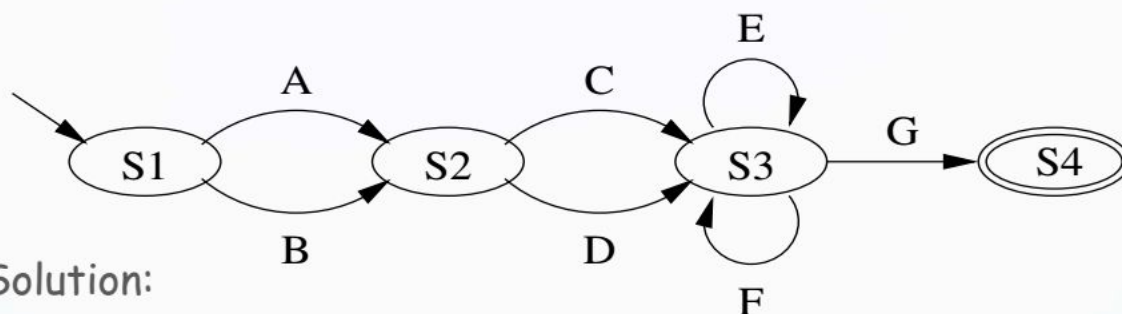
---

<sup>1</sup> Those paths can be split into control paths and data paths (the latter relating to the set of values the extended state can take miscellaneous model coverage criteria<sup>2</sup> are commonly used when designing a test suite with the help of a model.

<sup>2</sup> Bin99 Binder, R. V. Testing Object Oriented Systems; models, patterns, and tools. Addison-Wiley Longman Publishing Co. Inc. Boston, MA, USA, 1999

# Structural model coverage

## Transition-based



### Solution:

- All-states
  - A;C;G
- All-configurations
  - Equal to All-states
- All-transitions
  - A;C;E;F;G and B;D;G
- All-transition-pairs
  - Eg..at state S2: A;C, A;D, B;C, B;D
- All-loop-free-paths
  - A;C;G, A;D;G, B;C;G, B;D;G
- All-one-loop-paths
  - 4 paths of all-loop-free-paths + combination of each of these with a single loop around either E or F transition (4\*2\*4=12 tests)
- All-round-trips
  - A;C;E, A;C;F, A;C;G, A;D, B
- All-paths
  - 4 paths of all-loop-free-paths but extended with any number of E and F transitions

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Using a dedicated graph testing library, we computed the abstract test suite for the *All one-loop path* criteria and ended up with around 1.500 tests!! We reproduce below extract of the abstract test suite:

- A test is specified by a sequence of inputs
- Every line below is a sequence of control states the machine goes through based on the sequence of inputs it receives. Note that you can have repetition of control states, anytime a transition happens between a state and itself. Because we have used a *All one-loop path* criteria to enumerate the paths to test, every Team\_Detail loop corresponds to a different edge in the model graph. Here such loop transitions could be Skip Team or Join Team (valid form) or Join Team (invalid form). We can see from the extract how the graph search works (depth-first search).

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Team_Detail", "Team_Detail", "Team_Detail", "Teams", "Review", "State_Applied" ],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Team Detail", "Team Detail", "Teams", "Review", "State Applied"],
```

```
[["nok","INIT_S","Review","About","Review","Question","Review","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","State_Applied"],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Team Detail", "Team Detail", "Teams", "Review", "State Applied"],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Team Detail", "Teams", "Review", "State Applied"],
```

```
[["nok","INIT_S","Review","About","Review","Question","Review","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","State_Applied"],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Team Detail", "Team Detail", "Teams", "Review", "State Applied"],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Team_Detail", "Teams", "Review", "State Applied" ],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Team_Detail", "Teams", "Review", "State Applied"],
```

```
[ "nok", "INIT_S", "Review", "About", "Review", "Question", "Review", "Teams", "Review", "State Applied"],
```

```
["nok", "INIT S", "Review", "About", "Review", "Question", "Review", "State Applied"],
```

```
[["nok","INIT_S","Review","About","Review","Question","Question","Review","Teams",
,"Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","Sta
te Applied"],
```

...

```
["nok","INIT S","Review","State Applied"]
```

```
[ "nok", "INIT_S", "About", "Question", "Teams", "Team_Detail", "Team_Detail", "Team_Detail", "Team_Detail", "Teams", "Review", "About", "Review", "Question", "Review", "State Applied"],
```

```
[ "nok", "INIT_S", "About", "Question", "Teams", "Team_Detail", "Team_Detail", "Team_Detail", "Team_Detail", "Teams", "Review", "About", "Review", "Question", "Question", "Review", "State Applied"],
```

```
["nok","INIT_S","About","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","About","Review","State_Applied"],
...
["nok","INIT_S","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","About","Review","Question","Review","State_Applied"],
["nok","INIT_S","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","About","Review","Question","Question","Review","State_Applied"],
["nok","INIT_S","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","About","Review","State_Applied"],
["nok","INIT_S","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","About","About","Review","Question","Review","State_Applied"],
...(1000+ lines)
```

## Test Selection

As we mentioned, even for a relatively simple reactive system, we handed up with 1.000+ tests to exhaust the paths between initial state and terminal state, and that even with excluding n-loops.

We finally selected only 4 tests from the All path coverage set, for a total of around 50 transitions taken:

```
["nok","INIT_S","About","About","Question","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","Question","Review","About","Review","State_Applied"],
["nok","INIT_S","Question","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","State_Applied"],
["nok","INIT_S","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","State_Applied"],
["nok","INIT_S","Review","Teams","Team_Detail","Team_Detail","Team_Detail","Team_Detail","Teams","Review","State_Applied"]
```

Those tests :



- fulfill the *All transitions coverage* criteria: 4 input sequences are sufficient
- involves all the loops in the model graph (cf. first test sequence)
- insist slightly more on the core functionality of the system, which is to apply to volunteer teams (e.g. TEAM\_DETAIL loop transitions)
  - the transition space for that control state is the permutations of Join(Invalid Form) x Skip x Join(Valid Form), with  $|\text{set}| = 2$  for Join and Skip (an event triggering the associated transition happens or not). We have  $|\text{Join(Invalid Form)} \times \text{Skip} \times \text{Join(Valid Form)}| = 8$ , so  $3! \times 8 = 48$  transition permutations for that control state. Rather than exhaustively testing all permutations, we pick 4 of them, fit into the 4 input sequences that are necessary to cover the model.

In summary the process is :

- we have informal UI requirements which are refined into a state-machine-based detailed specification
- we generate input sequences and the corresponding output sequences, according to some model coverage criteria, our target confidence level and our testing priorities (happy path, error path, core scenario, etc.)
- we validate the selected tests manually

## Test Implementation

Once test sequences are chosen, test implementation is pretty straightforward. Because state transducers from our library are causal functions, i.e. function whose outputs depend exclusively on past inputs, it is enough to feed an freshly initialized state machine with a given sequence of inputs and validate the result sequence of outputs.

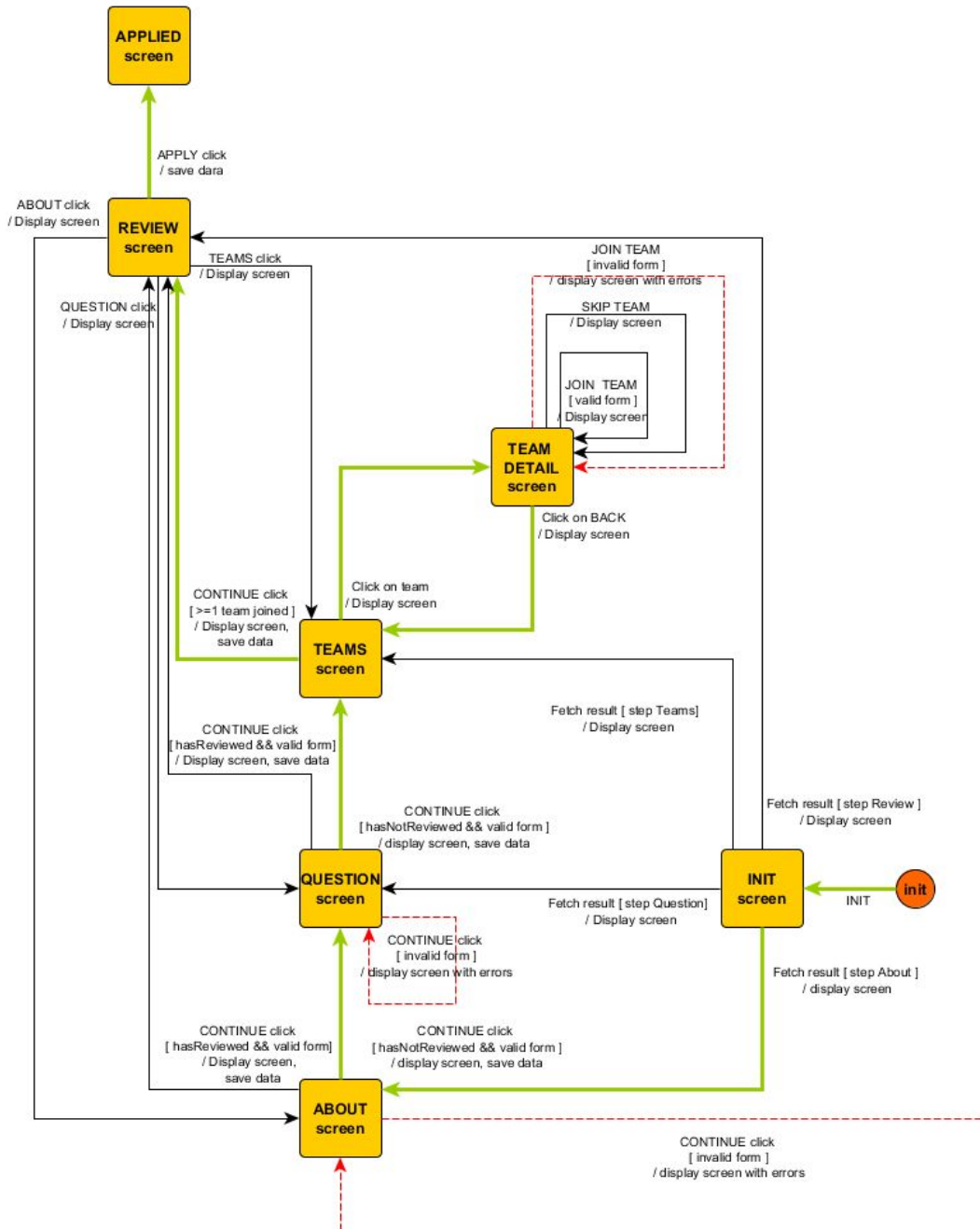
cf. test repository

## Integration Tests

Note that once the model is validated, we can use it as an oracle. This means for instance that we can take any input sequence, run it through the model, gather the resulting outputs, generate the corresponding BDD test, and run them. Most of this process can be automatized.

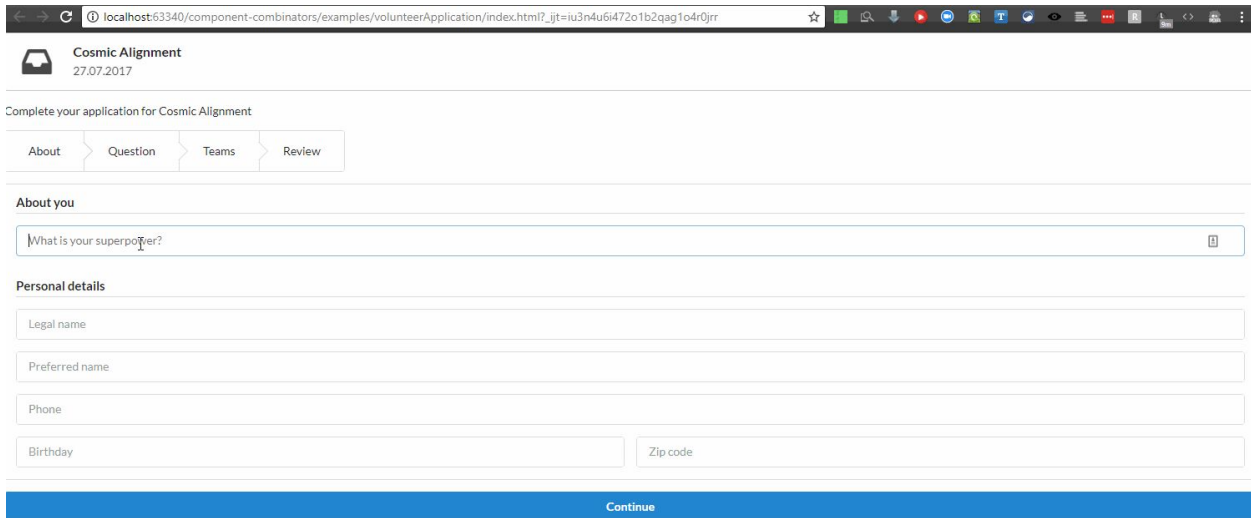
## Modelizing the User Flow With Extended State Machine

On the first iteration, the provided wireframes are refined into a workable state machine, which reproduces the provided user flow, while addressing key implementation details (error flows, data fetching).



The behaviour is pretty self-explanatory. The machine moves from its initial state to the fetch state which awaits for a fetch event carrying the fetched data (previously saved application data). From that, the sequence of screens flows in function of the user flow and rules defined.

Note that we could have included processing of the fetch event inside our state machine. We could have instead fetched the relevant data, and then start the state machine with an initial INIT event which carries the fetched data. Another option is also to start the state machine with an initial extended state which includes the fetched data.



The screenshot shows a web browser window with the address bar displaying `localhost:63340/component-combinators/examples/volunteerApplication/index.html?_ijt=iu3n4u6i472o1b2qag1o4r0jrr`. The page title is "Cosmic Alignment" with a date "27.07.2017". The main heading is "Complete your application for Cosmic Alignment". Below this is a navigation bar with four tabs: "About", "Question", "Teams", and "Review". The "About" tab is active. The form is divided into two sections: "About you" and "Personal details". The "About you" section has a text input field with the placeholder text "What is your superpower?". The "Personal details" section has four input fields: "Legal name", "Preferred name", "Phone", and "Birthday". The "Birthday" field is a date picker. There is also a "Zip code" field. At the bottom of the form is a blue button labeled "Continue".

## Running the Project on Your Local Machine

Check out the branch on your machine using command line or terminal and navigate into the correct directory using

```
git clone https://github.com/anishLearnsToCode/interface-design-automata.git  
cd interface-design-automata
```

Then to run the application

```
npm run start
```

Open your web browser (preferably Chrome or Mozilla) at port 8000.

See on browser at [localhost:8000/](http://localhost:8000/)

## Examples

### About Section

This is the first section of the form that asks basic information such as Hobbies, name, age, gender etc. If this section isn't filled completely then the form will not allow you to move forward with the application and the user will be stuck here as dictated by the transducer diagram.

The screenshot shows a web browser window with the address bar at localhost:8000. The page title is 'Cosmic Alignment' with a date of 27.07.2017. Below the title, it says 'Complete your application for Cosmic Alignment'. There are four tabs: 'About', 'Question', 'Teams', and 'Review'. The 'About' tab is active. Under 'About you', there is a text input field containing 'reading'. Under 'Personal details', there are four input fields: 'anish sachdeva', 'anish', '8287428181', and '7th april 1998'. To the right of the date field is a small input field containing '110034'. At the bottom of the form is a blue button labeled 'Continue'.

### Questions Section

In the questions section the organization is asking a generic question on how the new participant can contribute to the organization. The user can move forward with the application only once the user has filled in this section.

If the user doesn't answer the question, the control flow will not allow the user to move ahead with the application and will be stuck at the same state for an indeterminate amount of time.

anishLeamsToCode/interface-di x Cycle App x +

localhost:8000

CSRanking RegEx Class Central MS Learn levels oTentoms PirateBay S fe toc dbms cn dbms DTU DBMS GRE Calculus Probability Math Poems French PhD Physics NLP

**Cosmic Alignment**  
27.07.2017

Complete your application for Cosmic Alignment

About Question Teams Review

Organizer's question

**What good can you bring to Cosmic Alignment?**  
Organizer's name/role

I am dedicated

Continue

## Teams Section

The teams section has many different teams for which the user can select and fill out answers to. Each team has its own requirements and questions as a selection criteria and the user can answer selection questions to as many teams as he/she wishes, but must answer at least one question for one team.

The imaginary organization can then review these applications and assign teams to volunteers on the basis of their answers.

anishLeamsToCode/interface-di x Cycle App x +

localhost:8000

CSRanking RegEx Class Central MS Learn levels oTentoms PirateBay S fe toc dbms cn dbms DTU DBMS GRE Calculus Probability Math Poems French PhD Physics NLP

**Cosmic Alignment**  
27.07.2017

Complete your application for Cosmic Alignment

About Question Teams Review

Back to teams

**Box Office** The gateway to Cosmic Alignment! Here, you'll be selling GA tickets, banding guests and checking in artists. This job requires organization and a friendly face.

**Which festivals have you worked Box Office before?**  
Team lead's name/role

Please enter your answer here

Mandatory field : please fill in!

Skip this team or Join team

Cosmic Alignment  
27.07.2017

Complete your application for Cosmic Alignment

About Question Teams Review

Back to teams

Box Office The gateway to Cosmic Alignment! Here, you'll be selling GA tickets, banding guests and checking in artists. This job requires organization and a friendly face.

Which festivals have you worked Box Office before?  
Team lead's name/role

I love the cinema and would work efficiently

Mandatory field : please fill in !

Skip this team Join team

Cosmic Alignment  
27.07.2017

Complete your application for Cosmic Alignment

About Question Teams Review

Select a team

Parking	X
Box Office	X
Safety	O
Site Operations	X
Headquarters	X
Impact Reduction	X
Stage Hands	X
Build	X

Continue

## Review Section

Once, all the three sections - about, question and teams have been completed the review section is presented in front of the user.

This section is there just to review all the information that the user has entered and if he/she wishes to change some information he/she can click on the edit buttons and then follow the same procedure for any given section.

The review section also has the final application button which once pressed sends the data to the server and also stores the information entered by the user on the machine. If the user opens this application after closing it he/she will find that all his/her information has been retained.

anishLeamsToCode/Interface-di- Cycle App

localhost:8000

CSRanking RegEx Class Central MS Learn levels eToroTrents PirateBay S fe toc dbms cn dbms DTU DBMS GRE Calculus Probability Math Poems French PhD Physics NLP

Cosmic Alignment

27.07.2017

Does this look good?

About

Question

Teams

Review

About you

i reading

anish sachdeva (anish)

8287428181

27.07.2017

i 110094

Organizer's question

What good can you bring to Cosmic Alignment?

Organizer name and role

I am dedicated

Team selection

i Safety

i Impact Reduction

Apply for the things



## Conclusion

We have seen above through the examples and also the extensive testing regiment that this new mechanism works as advertised and does provide a way for developers to define user workflow as a finite automaton complete with stages, current state and also *actions* that will be triggered based on the information and command that a user enters at any given state of the application.

The proposed method also pushes the developer to think about her application in states rather than different functions and pages that might be triggered individually from any given point in the codebase.

Using the new mechanism the developer will think about the user interface at a much abstracted level and will directly think about how the user will interact and how different interactions will take the user to different states/pages.

And, then the developer can go about designing these pages directly without having to worry about how data will be interchanged between pages and managing control flow.

The mechanism provides the developer automatically with pipes that will transfer the data from one state to another when states change and will also automatically manage the page changes and user experience flow based on how the developer has syntactically defined it.

Hence, we can see that this new mechanism can readily speed up development and prototyping and brings the creation step further close to the design and user experience step by encapsulating the data piping and plumbing and routing under an abstracted design that requires the developer to only enter the finite state user flow diagram.

## Bibliography

1. [Git](#)
2. [GitHub](#)
3. [Angular](#)
4. [Node](#)
5. [Npm](#)
6. [interface-design-automata](#) [GitHub - Repository]
7. [Introduction to Theory of Computing](#) [MIT Press]
8. [Angular Hosting on Github Pages](#) [Telerik]
9. [gh-pages](#) [npm]
10. [Introduction to Theory of Computation](#) by Michael Sipser [Cengage Learning]
11. [Deterministic Finite State Automaton](#) [Wikipedia]
12. [Clean Code](#) by Robert C. Martin (Uncle Bob) [Amazon]
13. [GraphViz](#)
14. [State Transducer](#) [Wikipedia]
15. [State Transducer](#) [cs-union]