StateMap
A DFA Simulation Language

Oren Finard
Jackson Foley
Alexander Peters
Brian Yamamoto
Zuokun Yu
Motivation

- Easily converts a DFA drawing into code.

- Applications include regex matching and event-based code (ex. CYOA)

- Allows multiple DFAs to take concurrent steps and interact.

- Applications in hardware with synchronous components and clocks.
// Prints hello world ten times.
void DFA main( /* args */)
{
    int count = 0;

    start
    {
        hello <- count < 10;
        finished <- *;
    }

    hello
    {
        print("Hello World!");
        count = count + 1;
        start <- *;
    }

    finished
    {
        return;
    }
}

// Sub-DFAs must be declared above the main DFA. If a DFA calls a sub-DFAs,
// waits for the sub-DFA to return.
int DFA sum(int a, int b)
{
    start
    {
        return (a+b);
    }
}
void DFA main()
{
    start
    {
        int res = sum(1,2);
        printf(res);
        return;
    }
}
// Prints hello world ten times.
void DFA main(/* args */)
{
    int count = 0;

    start
    {
        hello <- count < 10;
        finished <- *;
    }

    hello
    {
        print("Hello World!");
        count = count + 1;
        start <- *;
    }

    finished
    {
        return;
    }
}
(Sub-DFAs must be declared above the main DFA - if a DFA calls a sub-DFAs, waits for the sub-DFA to return.)

```c
int DFA sum(int a, int b) {
    start {
        return (a+b);
    }
}

void DFA main() {
    start {
        int res = sum(1,2);
        print(itos(res));
        return;
    }
}
```
How to compile

1) make
2) ./compiler
   [name of output] < [path to .sm file]
3) python [name of output].py [args]
Matching strings against Reg Exs

(ab|c*)d*

Note: "string" will pass the program the argument [s.t.r.i.n.g]

Note to the note: That's a double quote followed by a single quote.
against Reg Exs

\((ab|c^*)d^*\)
/* A StateMap DFA that accepts the regex \( (ab|c^*)d^* \) */

```cpp
void DFA main(stack<string> args) {
    int accepted = 1; /* acceptance state if reach end of stack */

    start {
        string s = args.peek();
    }

    stateOne {
        accepted = 0;
        args.pop();
        string s = args.peek();
        stateFour <- s == "b";
        notAccept <- *
    }

    stateTwo {
        accepted = 1;
        args.pop();
        string s = args.peek();
        stateThree <- s == "d";
        stateTwo <- s == "c";
        accept <- s == EOS;
        notAccept <- *
    }
}
```
Note: "'string'" will pass the program the argument [s,t,r,i,n,g]

Note to the note: That's a double quote followed by a single quote.

```
bky2102@cairo:/StateMap/StateMap$ ./compiler < sample_programs/reg_ex_test.sm
bky2102@cairo:/StateMap/StateMap$ python output.py "'aabbcc'"
Not accepted by the DFA
bky2102@cairo:/StateMap/StateMap$ python output.py "'abddd'"
Accepted by the DFA.
```
Begin serving, V5 So, the next thing you’re going to do in this class is not just drop in like I did for the rest of the class. This is aiks for a semester-long TDA project. This is a team programming project. Now, the team part of it is exactly the worst, most difficult aspect of all of it. But just to make it more difficult, I’m going to make you design and implement your own language and a compiler for it... you’re going to have to work with other human beings. This really sucks. (V5)

At this point, you’re strongly considering passing up and leaving. You don’t know anyone in this class and stopping in people’s desks did not make a great first impression on your prospective classmates. As you walk through the rest of the lecture and escape, you find yourself in the hallway. You notice the door open and your curiosity gets the best of you. You step inside...

Choose Your Own Adventure

You’re in a decision:

1. Stay in the course and knock it out.
2. Drop out of the course.

[Diagram of a decision tree with paths and outcomes]
beginning to feel uncomfortable about this class. You dread public speaking. You dread massive, final reports.

print("He continued, \"So, the main thing you're going to do in this class, assuming that you don't drop it like I want you to, is a semester long TEAM project. This is a team programming project. Now, the team part of it is easily the worst, most difficult aspect of all of it, but just to make it more difficult, I'm going to make you design and implement your own language and a compiler for it... you're going to have to work with other human beings. This really sucks.\"\n");

print("At this point, you're strongly considering packing up and leaving. You don't know anyone in this class and stepping on people's desk did not make a great first impression on your prospective classmates. As you sit through the rest of the lecture and attempt to focus on Professor Edwards' words through your rising panic, you've come to a decision.\n");

print("You've decided to: ");

print("1) Stay in the course and tough it out.");
print("2) Drop out of the course. ");
choice = input("\n");

dropkicked <- choice == "2";
randomGroup <- choice == "1";
print("Type 1 or 2 to indicate your choice.\n");
sittingInTheBackSeat <- *;
Non-Standard Features

Control Flow Expressed Through Transitions
- If, While, For
  if (x == 4) {
    // do stuff
  }
  state {
    end <- x != 4;
    // do stuff
  }

while (x < 4) {
  // do stuff
}
state1{
  state2 <- x < 4;
  end <- *;
}
state 2 {
  // do work
  x = x + 1;
  state1 <- *;
}

Does this work?
state1{
  state1 <- x < 4;
}

Limited primitives:
- String
- Float
- Int
- Stack<"Primitive“>

Mock concurrency
Step through DFAs one state at a time.
Envision a debugger.
Control Flow Expressed Through Transitions

- If, While, For

```python
if (x == 4) {
    // do stuff
}

state {
    end <- x != 4;
    // do stuff
}
```
while (x < 4) {
  // do stuff
}
state1{
  state2 <- x < 4;
  end <- *;
}
state 2 {
  // do work
  x = x + 1;
  state1 <- *
}
Does this work?
state1{
    state1 <- x < 4;
}

Limited primitives:
- String
- Float
- Int
- Stack<"Primitive">

Mock concurrency
Step through DFAs one state at a time.
Envision a debugger.
Architecture
StateMap Source -> Scanner -> Tokens -> Parser -> Abstract Syntax Tree -> Semantic Check -> Code Generation -> Python!
int DFA get_int() {
    start {
        print("1");
        finish <- *;
    }

    finish {
        print("2");
        return 1;
    }
}

int DFA hello() {
    start {
        print("hello");
        finish <- *;
    }

    finish {
        print("world");
        return 1;
    }
}

void DFA main() {
    start {
        concurrent(get_int(), hello());
        return;
    }
}
```python
class _main:
    _now = _node_start
    def __init__(self,*args):
        try:
            pass
        except IndexError:
            print('RuntimeError:Too few arguments provided to dfa "main"')
            sys.exit(1)
        self._returnVal = None
        _main._now = self._node_start
        self._next = None
    while self._returnVal is None:
        _main._now()
        _main._now = self._next
        return
    def _node_start(self):
        concurrent(_get_int, [], _hello, [])
        self._returnVal = 1
        self._next = None
_dfa_Dict["main"] = _main

class _hello:
    _now = _node_start
    def __init__(self,*args):
        self._returnVal = None
        _hello._now = self._node_start
        self._next = None
    return
    def _node_finish(self):
        print "world"
        self._returnVal = 1
        self._next = None
    if(1):
        self._next = self._node_finish
        return
_dfa_Dict["hello"] = _hello

class _get_int:
    _now = _node_start
    def __init__(self,*args):
        self._returnVal = None
        _get_int._now = self._node_start
        self._next = None
        return
    def _node_finish(self):
        print "2"
        self._returnVal = 1
        self._next = None
    def _node_start(self):
        print "1"
        if(1):
            self._next = self._node_finish
            return
_dfa_Dict["get_int"] = _get_int
```
Concurrent in Python

```python
def concurrent(dfaArgs):
    dfaargs = [dfa for i, dfa in enumerate(dfaArgs[1:])]
    finishedDfAs = set()
    while len(set([dfa._class__new() for dfa in dfaArgs]) - finishedDfAs):
        for dfa in (set(dfaArgs[1:]) - finishedDfAs):
            dfa._class__new()
            if dfa not in finishedDfAs:
                finishedDfAs.add(dfa)
    return set(dfa._returnVal for dfa in dfaArgs)
```

Call instance method of each class variable

Initialize instance of classes
[df1_pointer, df1_args, df2_pointer, df2_args, ...]
(takes a variable number of them)

```python
def concurrent(*dfasNArgs):
    dfas = [dfa(dfasNArgs[i*2+1]) for i, dfa in enumerate(dfasNArgs[::2])]
    finishedDfas = set()
    while len(set(dfas) - finishedDfas):
        for dfa in (set(dfas) - finishedDfas):
            dfa.__class__.__now()
        for dfa in (set(dfas) - finishedDfas):
            dfa.__class__.__now = dfa.__next
        finishedDfas = set([dfa for dfa in dfas if dfa.__returnVal is not None])
    return str(dfa.__returnVal) for dfa in dfas
```

- Instantiates the instances of the dfas, and puts them in a list
- Iterate over dfas that haven't finished
- Set class variable as instance method
- Call instance method from class variable
- Return stack of strings
### Shift Register

<table>
<thead>
<tr>
<th>Clock</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>D1</td>
<td>D1</td>
<td>Read</td>
<td>D2</td>
<td>D2</td>
<td>Read</td>
<td>D3</td>
<td>D3</td>
<td>Read</td>
<td>D4</td>
<td>D4</td>
<td>Read</td>
<td>D5</td>
<td>D5</td>
<td>Read</td>
<td>D6</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Read</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
</tr>
</tbody>
</table>

### Diagrams

**CLOCK**
- Low:
  - Sleep
- Rising:
  - * (Transition)

**DATA**
- Read:
  - Counter = 0
  - Counter == 1
  - EOS OR "0"
- Error:
  - ++Counter

**DFF**
- Clock Rising AND Prev High
- Clock Rising AND Prev Low
- High:
  - Sleep
- Low:
  - Sleep

The diagrams represent the state transitions and conditions for the shift register, including clock and data states, along with decision points for the counter and error states.
// DFA to represent a clock
// halfPeriod: integer to represent period/2 in ms
void DFA clock(int halfPeriod)
{
    // Start == low
    // Wait halfPeriod ms, then toggle
    start
    {
        sleep(halfPeriod);
        rising <- *;
    }

    // state that triggers a catch for the DFFs
    rising
    {
        high <- *;
    }

    high
    {
        sleep(halfPeriod);
        start <- *;
    }
}
Shift Register

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>L</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Read</td>
<td>D1</td>
<td>D1</td>
<td>Read</td>
<td>D2</td>
<td>D2</td>
<td>Read</td>
<td>D3</td>
<td>D3</td>
<td>Read</td>
<td>D4</td>
<td>D4</td>
<td>Read</td>
<td>D5</td>
<td>D5</td>
<td>Read</td>
<td>D6</td>
</tr>
<tr>
<td>DFF1</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
<td>D4</td>
<td>D4</td>
<td>D4</td>
<td>D5</td>
<td>D5</td>
</tr>
<tr>
<td>DFF2</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
<td>D4</td>
<td>D4</td>
<td>D4</td>
</tr>
<tr>
<td>DFF3</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
</tr>
<tr>
<td>DFF4</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
</tr>
<tr>
<td>Display</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>read</td>
<td>read</td>
<td></td>
</tr>
</tbody>
</table>

CLOCK

- low (sleep)
- rising
- high

- * clock rising AND prev high

DATA

- high (sleep)
- read
- counter = 0
- counter = 1
- counter == 1
- counter == 1
- low (++counter)
- error

- * "1"
- * EOS OR "0"

DFF

- low (clock rising AND prev low)
- high
// 1st T-FlipFlop in Shift Register  
// Catches data on every rising clock  
void DFA DFF1()
{
    // low output  
    start  
    {
        high     <- (state("clock") == "rising" && state("dataIn") == "high");  
        start    <- *;  
    }

    // high output  
    high  
    {
        start     <- (state("clock") == "rising" && state("dataIn") == "low");  
        high      <- *;  
    }
}
### Shift Register

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td>H</td>
<td>L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Read</td>
<td>D1</td>
<td>D1</td>
<td>Read</td>
<td>D2</td>
<td>D2</td>
<td>Read</td>
<td>D3</td>
<td>D3</td>
<td>Read</td>
<td>D4</td>
<td>D4</td>
<td>Read</td>
<td>D5</td>
<td>D5</td>
<td>Read</td>
<td>D6</td>
</tr>
<tr>
<td>DFF1</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
<td>D4</td>
<td>D4</td>
<td>D4</td>
<td>D5</td>
<td>D5</td>
</tr>
<tr>
<td>DFF2</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
<td>D3</td>
<td>D4</td>
<td>D4</td>
</tr>
<tr>
<td>DFF3</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
<td>D3</td>
<td>D3</td>
</tr>
<tr>
<td>DFF4</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>D1</td>
<td>D1</td>
<td>D1</td>
<td>D2</td>
<td>D2</td>
<td>D2</td>
</tr>
<tr>
<td>Display</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td>read</td>
<td>print</td>
<td>start</td>
<td></td>
</tr>
</tbody>
</table>

**CLOCK**

- **low**
  - sleep

- **rising**
  - *

- **high**
  - *

**DATA**

- **read**
  - counter = 0
  - counter == 1
  - counter == 1

- **error**
  - "1"
  - ++counter

- **high**
  - sleep

- **low**
  - ++counter

**DFF**

- **clock rising AND prev high**
  - *

- **clock rising AND prev low**
  - *
void DFA dataIn(stack<string> data) {
    int counter = 0;
    // Read state
    start {
        counter = 0;
        low <- data.peek() == EOS;
        string currData = data.pop();
        high <- currData == "1";
        low <- currData == "0";
        error <- *;
    }
    low <- *;
    start <- counter == 1;
    counter = counter + 1;
    error <- *;
}

// high and low states to represent the current data input. // counter is used for synchronicity
high {
    start <- counter == 1;
    counter = counter + 1;
    high <- *
}
Testing in Progress ...
Lessons Learned

- Tests are king
- #gitblame
- Everyone has strengths (and weaknesses)
- Copying is good, thinking is better, and OCAML isn't actually that bad.
StateMap
A DFA Simulation Language

Oren Finard
Jackson Foley
Alexander Peters
Brian Yamamoto
Zuokun Yu