SAKÉ

Shalva Kohen: sak2232 (Language Guru)
Arunavha Chanda: ac3806 (Manager)
Kai-Zhan Lee: kl2792 (System Architect)
Emma Etherington: ele2116 (Tester)

Introduction:

Behind all models of computation and computer science lies the concept of automata, or Finite State Machines (FSMs). However, many of the problems that arise in Computer Science Theory are considered unsolvable, including:

- Reachability
- State Minimization
- Equivalence

We plan to design a language that can manipulate and simulate FSMs, while also answering complex algorithmic questions such as reachability. Our solution is Saké, the name being derived from the first letters of our team members: **S**halva, **A**runavha, **K**ai-Zhan, **E**mma.

Language Description:

Saké is a simple language designed to manipulate, simulate, and perform reachability tests on concurrent finite state machines (FSM). These include the Deterministic Finite Automata (DFA), Nondeterministic Finite Automata (NFA), Moore, and Mealy machines. Given a specific input length or otherwise-specified constraint, the user will be able to test for state reachability in an FSM system, a single FSM or multiple concurrent FSMs. The user will also be able to simulate an FSM system on a given input.

Our language is bipartite: it consists the description of the FSM that the user desires to work with - called “Bottle” - and of the actions that the user wants performed on the FSM, the main Saké code.

The Bottle code may be located either in a file outside of the Sake code or in the same file as the rest of the code, much like header files in C. Within the Bottle code, the user would also be able to elaborate on any actions the FSM should perform in a specific state. The user will be able to choose four modes with which to describe their FSM. The general syntax for all four modes would be the same. However, there would be small differences in the descriptions. For example, if the user specifies that they want to write a Moore Machine, they would have to indicate what outputs each state has. If the user chooses to define a DFA, then they would also have to determine an end state within that system.

The second aspect of our language focuses on the actions performed on the FSM. This part of our language will be less descriptive, and more algorithmic. The main function of Saké code is called fill(). If the names are intuitively tough to understand, remember: *with Saké, we fill() the Bottle!*

General Syntax:

Types:

- empty (0 byte)
- byte (1 byte)
- int (4 byte)
- long (8 byte)
- float (8 byte)
- array (n * sizeof(type) byte)

Macros:

- string: #define string(name, value) (char[len(value)] name = value)
- char: #define char byte
- bool: #define bool byte

Operators:

Operator	Description
=	Assignment operator
+, -, *, /	Arithmetic operators
( ), { }	Grouping: parentheses for expressions, curly braces for statements
+=, -=, *=, /=	Arithmetic assignment operators
.	The dot operator
[ ]	Square brackets
&&, ||, !	And, or, not
<, <=, ==, >, =>	Relational and equality comparisons

Keywords:

Keyword	Description
moore, mealy, dfa, nfa	The type of FSM
state	To define a state of the FSM
start	Assigning a start state
depends	Adding dependencies for concurrent state machines
input	Defining sequence of inputs for FSM
link	Defining state transitions
end	Assigning an end state
actions	Tasks performed in a specific state of the FSM
if, elif, else	If-else ladder statements
while	While loop
for	For loop
continue	Restart execution of loop at beginning of scope
break	Exit out of loop
return	Return statement
import	Import an FSM
as	Renaming as an alias
ep	Epsilon transition for NFA

Comments:

- `~ This is a line comment`
- `/~ This is a block comment ~/`

FSM Functions:

Functions	Description
sim(input)	Simulates the given FSM on an input received from standard in. Returns the output of the FSM
is_reachable(<test state> , <constraint>)	Tests for reachability of a given state within a certain number of steps in an FSM. Returns a boolean. Will return an error if the test state is not defined in the FSM.

Control flow:

If Statements:

```
if condition {
      /~ code block ~/
}

if condition {
      /~ code block ~/
} elif {
      /~ code block ~/
} else {
      /~ code block ~/
}
```

While Loop:

```
while condition {
      /~ code block ~/
}
```

For Loops:

```
for s in states {
      /~ code block ~/
}

for s in [start,end,interval] {
      /~ code block ~/
}
```

FSM Declaration:

FSM Declaration:

```
<fsm_type_keyword> <fsm_name> {
    /~
     ~ Specification of input, outputs, states, and
     ~ other fsm properties
     ~/
}
```

Concurrent FSM Declaration:

```
<fsm_type_keyword> <fsm_name> depends <fsm_name> {
    /~ fsm specification ~/
}
```

Concurrent FSM Declaration with Aliasing:

```
<fsm_type_keyword> <fsm_name> depends <fsm_name> as <alias> {
    /~ fsm specification ~/
}
```

Multiple Concurrent FSM Declaration:

```
<fsm_type_keyword> <fsm_name> depends <fsm_name>, <fsm_name> {
    /~ fsm specification ~/
}
```

Multiple Concurrent FSM Declaration with Aliasing:

```
<fsm_type_keyword> <fsm_name> depends
    <fsm_name> as <alias>, <fsm_name> as <alias> {
    /~ fsm specification ~/
}
```

Input/Output Declaration:

```
input [<input_type> <input_name> , <input_type> <input_name> ,...]
output [<output_type> <output_name> ,...]
```

State Declaration:

Explicit state declaration:

```
state <state_name>, <state_name>, ...
```

Shorthand declaration (start and stop inclusive) :

```
state <prefix>[<start_number> - <stop_number>]
```

State Definition :

Moore:

```
<state_name> (<state_output>, ...) {
    /~
     ~ Specification of transitions, and other state properties
     ~/
}
```

Mealy, DFA, NFA :

```
<state_name> {
    /~ state specification ~/
}
```

Start State Definition :

```
start <name_of_start_state>
```

Transition Declaration:

Mealy:

```
link <transition_dest>(<input>,...)
```

Moore, DFA, NFA :

```
link <transition_dest>(<input>,...)/(<output>,...)
```

Concurrent:

```
link <transition_dest>
    (<input>,...,[<concurrent_fsm_alias>.<state_name>,...]) /
    (<output>,...)
```

Actions Within A State:

```
state s {
	actions {
	/~ actions performed by the fsm within the current state ~/
	}
}
```

Main Function Declaration:

```
int fill() {
	/~ code block ~/
	return <return_value>
}
```

Helper Function Declaration:

```
<return_type> <function_name>(<parameters>) {
	/~ code block ~/
	return <return_value>
}
```

Sample Code:

Bottle Code For A Moore Machine:

```
moore example_fsm {
      input [char char_input, int int_input]
      output [int a, int b]

      state s[0-1]
      start s0

      s0 (1,0) {
            link s0('a', 1), s1('b', 2)

            actions {
                  /~ specification for maintaining state but actions ~/
                  e.g
                  if char_input == 'w' {
                        /~ Perform some task ~/
                  }
            }
      }

      s1 (0,1) {
            link s0('b', 2), s1('b', 1)
      }
}
```

Bottle Code For A Mealy Machine:

```
mealy example_fsm {
      input [char char_input_name, int int_input_name]
      output [int a, int b]

      state s[0-2]
      start s0

      s0 {
            link s0('a', 7)/(1, 0), s1('a', 7)/(1, 1), s2('a', 7)/(0, 0)
      }

      s1 {
            link s0('a', 9)/(0, 0), s2('b', 7)/(1, 0)
      }

      s2 {
            link s0('a', 5)/(1, 0), s1('a', 7)/(0, 1), s2('c', 7)/(1, 0)
      }
}
```

Bottle Code For Concurrent FSMs:

```
moore example_fsm depends conc_other as c {
      input [char char_input_name, int int_input_name]
      output [int a, int b]

      state s[0-1]
      start s0

      s0 (1,0) {
            link s1('a', c.x1)/(1)
            link s3('a', c.x2)/(0)
      }

      s1 {
            link s1('b', 2)
      }
}
```

Saké Code:

```
import example_fsm as f1    ~ located in Bottle file "example_fsm.bl"

empty fill() {
      print "Simulating f1 on c9w2: " + f1.sim("c9w2")
      print "Enter text to simulate f1 on: " + f1.sim(stdin)
      print "Reachability of state s2 in 4 steps: "+
            f1.is_reachable(s2, 4)
}
```