SAKÉ LRM

Shalva Kohen: sak2232 (Language Guru)

Arunavha Chanda: ac3806 (Manager)

Kai-Zhan Lee: kl2792 (System Architect)

Emma Etherington: ele2116 (Tester)

Contents:

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

Saké is a simple language designed to describe, simulate, and perform reachability tests on concurrent Moore finite state machines (FSM). 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. Within the Bottle code, the user would also be able to elaborate on any actions the FSM should perform in a specific state. 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!*

Language Basics:

Types:

The user will be able to use certain primitive types to define their variables, inputs, and outputs of the fsm. These types are:

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

As indicated, an int will be 4 bytes long and represents a non-decimal number. A long will be 8 bytes long and is a very large non-decimal integer. Float types will be 8 bytes long of a decimal number. Finally, the user will be able to represent multiple elements of any type by using an array. In addition to these types, the user can also specify a value that is 0 bytes long by using the keyword **void**. One example of this is when a function does not return a value.

Other non-primitive types included in our language are:

- char (1 byte)
- string: (array of chars)
- bool: (1 byte)

A char is a 1-byte character defined by ASCII. A string is an immutable array of chars with an arbitrary length. A bool is a logical true or false value. The user will also be able to specify their own types using the **type** keyword.

Identifiers, Declaration, and Casting:

Variable identifiers, or names, begin with an underscore or a lowercase letter; the characters afterwards may consist of uppercase or lowercase letters, digits, and underscores. A formal regex for a variable name is the following:

```
[a-z_][a-zA-Z0-9_]*
```

Enum value identifiers, which will be discussed in greater depth, begin with a capital letter; the characters after follow the same patterns as in variable names:

```
[A-Z][a-zA-Z0-9_]*
```

Variables are implicitly typed; there are four primitive types, and users may declare other types based on these four: **bool**, **char**, **int**, and **float**. Variables are declared by assignment:

```
<variable> = <expression>
```

Literals:

Type	Regex (extended)	Description
bool	`True|False`	A bool literal is represented using the values "True" and "False". These names are capitalized to prohibit variable names from taking their stead. `True || False`
char	`'\\?[a-z]'`	A character is 1-byte long representing an ASCII symbol.
int	`[0-9]+` (with some limits)	Int literals are notated in base 10 by a series of digits. An integer that is outside the inclusive bounds of -2^31 and 2^31 - 1 will raise an error. Int literals are always positive and can be negated by a unary negative sign. `0 00 01 02 3562`
float	`[0-9]+\.[0-9]*|\.[0-9]+`	Float literals are notated by digits followed by a decimal point, optionally followed by more digits, or a decimal point followed by digits. `10.33 .67`

string	".*"	A string literal is exactly equivalent to an array literal declaration. However, it uses a more succinct form: its start and end are marked with double-quotations, and the characters that comprise the array are located between. Escaping works in strings as it does in characters. “Hello World!\n”
array	[<expr>, ...]	An array literal is specified by a comma-separated list of expressions in brackets that evaluate to the same type. Arrays may be multi-dimensional, but they may not be jagged. [‘a’, ‘b’, ‘c’] [[1, 2], [3, 4], [4, 5]]

Type Definition:

Custom types can be defined with the `type` keyword. Type identifiers are identical to variable identifiers. Their values may consist either of enum values or of expressions that all evaluate to a single type.

```
type <name> = <value> | <value> …

type bit = 0 | 1
type abc = ‘a’ | ‘b’ | ‘c’
type color = Red | Green | Blue | Yellow | Magenta
```

Casting:

In the event that one expression must be converted to another type, one may cast that type by calling the typename on the expression. This type conversion may not performed to or from arrays nor enums. The boolean converter simply checks if the input is equal to 0. Floats are truncated when casted to integer types, and a primitive may be cast to a custom-defined type whose values are limited to the primitive's type.

```
<variable> = <type>(<expression>)
```

```
three = int(3.14)
```

Operators:

Operator	Description
=	Used to define a new variable and set its value, or to set the value of a previously-defined variable. Variables are implicitly typed. `<variable> = <value>`

+, -, *, /	Arithmetic operators for adding any of the primitive types, or any of the custom types based on the primitive types; in the case that arithmetic is performed on custom types, the resulting values will be of the original primitive type. In addition, + and - are unary operators that respectively do nothing to and negate their operands.
	`type bit = 0 | 1`
	`bit(1) + bit(0) ~ returns an int`
	`1 + 2`
	`- 3`

+=, -=, *=, /=	Arithmetic assignment operators that are equivalent to assignment to the current value with the arithmetic operation specified.
	`<variable> += <value>`
	`<variable> = <variable> + <value> ~ equivalent`

|	Or operators that is used to create groups of types and states.
	`type <new_type> = <value1> | <value2> | ...`
	`state <name1> | <name2> | ...`

.	Resolution. Used to call an fsm function on an fsm.
	
	`<fsm_name>.sim(<input>)`
	`<fsm_name>.tick(<input>)`
	`<fsm_name>.reset()`

[ ]	Used for declarations of arrays of fsms, each with unique behavior depending on their index in the array. Also used for declarations of arrays of types for variables.
	
	`fsm <fsm_name>[<variable>] { … }`
	`<type>[<int expression>] <variable>`
	
	Also used to specify input and output streams in FSM definitions:
	
	`input [<type> <variable name>, ...]`
	`output [<type> <variable name>, ...]`

&&, ||, !	Boolean operators for evaluating boolean expressions; these symbols represent the and, or, not operators, respectively. Any integer value that is non-zero is treated as “true”, and any 0-value is treated as “false”. These expressions evaluate to either 0 or 1. Boolean arithmetic with these operators can only be performed on integers. `1 && 2 || !3`
<, <=, ==, !=, >, =>	Relational and equality comparisons, which return integer values. `1 != 2 && 2 == 3`

Expressions:

Expression	Description
(<expr>), {<stmt>}	Grouping: parentheses for expressions, curly braces for statements

_	Value wildcard, to be used in switch statements to indicate acceptance of any value.
	
	`switch  {`
	`    case _ : goto state_destination`
	`}`

Keywords:

Keyword	Description
fsm	The user must specify the **fsm** keyword to indicate the start of a FSM declaration. `fsm <fsm_name> { /~ FSM Specifications ~/ }`
state	The user must specify the **state** keyword within the fsm declaration to indicate the set of states of the fsm. Groups of states may be specified using a Pythonic list declaration. `state <name1> | <name2> | <name3> | ...`

start	The user has the ability to specify the start state of the fsm by using the **start** keyword. If the user does not specify the start state, then the default start state is the first state specified.
	
	`start <name> (<state_outputs>) {`
	`      /~ Specifications ~/`
	`}`

input	The user must specify the form of the input stream to be interpreted by the fsm by using the **input** keyword followed by the input form in square brackets. `input [<type> <variable_name>, ...]`
output	The user must specify the form of the output of the fsm by using the **output** keyword followed by the input form in square brackets. `output [<type> <variable_name>, ...]`

type	The user can define new variable types or reuse old ones, as described above, by using the **type** keyword. The type specification is delineated with parallel bars.
	`type bit = 0 | 1`
	`type abc = ‘a’ | ‘b’ | ‘c’`
	`type color = Red | Green | Blue | Yellow`

if, elif, else	The keys words **if**, **elif**, and **else** can be used to create ladder statements that can be used to specify blocks of code that should only be evaluated if a condition evaluates to true.
while	The **while** keyword can be used to specify a loop that should repeatedly execute a block of code while the condition evaluates to true.
for	The **for** keyword can be used to specify a loop that should execute a finite number of times. It can either execute over a set of states or for a set number of times.
continue	The **continue** keyword can be used within a while or for loop to exit the current evaluation on the body of the loop and move on to the next iteration of the loop.
break	The **break** keyword can be used within a while or for loop to exit the loop.

switch, case	The **switch** keyword is used within the body of a state specification to define the transitions of each state in the fsm. The **switch** keyword starts the transition specification. If the transitions are dependent on a specific input or a concurrent fsm, these dependencies can be specified in parentheses directly after the switch keyword. The input specific dependencies are specified followed by the concurrent fsms dependencies. Within the specification the **case** keyword is used to specify specific transitions. After the case keyword, the user must specify first the input values followed by the name(s) of the state(s) the concurrent fsm must currently be in.
	
	switch(<input_specifc>, ..., <concurrent_fsm_names>, ...) {
	        case <value>, ..., <state_names> : **goto** state_destination
	}

goto	The **goto** keyword is used within the state transition specification to specify which state to transition to. **case** <value>, ..., <state_name>, ... : **goto** state_destination
sysin	The **sysin** keyword can be used in the fill() function to designates that input is coming from standard input.

Built-In Functions:

Functions	Description
<fsm_instance>.sim(<inputs>, ...,[<fsm_names>])	Simulates the given FSM on an input, maybe received from standard in (and an optional set of concurrent FSMs). Returns the output of the FSM
<fsm_instance>.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.
print(<type> input)	Prints the input.
<fsm_instance>.tick (int i)	can manually specify how many clock ticks the fsm should simulate on
<fsm_instance>.reset()	Resets the FSM to the start state

Comments:

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

Control flow:

If-else Statements:

If-else blocks can be used to check conditions before evaluating blocks of code. The code block under the if statement will only be executed if the condition evaluates to true. If there is an else statement, it will execute is none of the if statements evaluates to true. Else statements are not required after if statements.

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

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

While Loop:

A while loop can be used to repeatedly execute a block of code while the condition evaluates to true. If the condition evaluates to false the first time the while loop is met, then the block of code will never be executed.

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

For Loop:

A for loop can be used to execute a block of code a finite number of times. A for loop can be applied to iterate through a set of states, or to execute from a start value to an end value with a set interval.

```
/~ where <states> is the defined set of states to iterate over ~/
for s in <states> {
    /~ code block ~/
}

/~start, end, interval have to be integers ~/
for s in [start:end:interval] {
    /~ code block ~/
}
```

FSM Specific Rules:

Structure:

Our language is designed so the user can create both regular and concurrent FSM. The following structure is meant to be done all in one file. Each point will be elaborated on in the following sections.

1. (Optional) The user defines any types they think they might want to use
2. FSM declaration
3. Input/Output Declaration
4. State declaration
5. (Optional) Any variable assignments or other computation not pertaining to a single state
6. State definition
    a. Within each state define transitions and any actions the user wishes to perform in a single state
7. Simulate the fsm in the fill() function

FSM Declaration:

The user needs to specify the FSM they wish to create using the **fsm** keyword. Everything within the { } will pertain to the FSM.

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

Input/Output Declaration:

To specify the type and quantity of inputs within the FSM, the user should use the **input** keyword and place in brackets and type and name of their input. If they wish to have more than one input they can do so using a comma separated list. The same thing is done for output declaration. However, instead of using the keyword **input** they use **output**. This is an example of a basic input and output statement:

```
      input [int i , bool b]

      output [String s]
```

Or, if the user wishes to define their own type they can do so outside of the fsm declaration and then use that type for their input/output declaration.

```
type <type_name> = {<primitive> <value>, ...}

fsm <fsm_name> {

      input [<type_name> <input_name> ,...]

      output [<type_name> <output_name> ,...]
      ...

}
```

State Declaration:

An FSM needs states! The user must first declare what states they wish to have before assigning transitions and outputs to them. This can be done in two ways. In the first way, the user can declare the name of each individual state and separating them with a logical or (|). Or, the user can use a shorthand notation to declare a list of numerically labeled states.

Explicit state declaration:

```
        state <state_name> | <state_name> | ...
```

Shorthand declaration (start inclusive and stop exclusive) :

```
        state <prefix>[<start_number>:<stop_number>:<interval>]
```

note: if interval is not specified, it is set to a default value of 1

State Definition :

Once the states are declared, the user can specify the actions performed by that state. This is where all transitions and outputs are defined. State outputs are specified as a comma separated list right after the state name. Since the user declared the output types above, the compiler will check the state outputs and make sure the appropriate types and number of outputs are used.

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

Start State Definition :

An FSM needs a start state. This is done by putting the keyword **start** before a state definition. The compiler will throw an error if there is not exactly one start state.

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

Transition Declaration:

Within each state definition, the user can specify transitions. There are several ways to do this. The first way uses a switch case approach. This approach works only if the user is defining transitions on inputs. After the keyword **switch** the user specifies which input variable the statement should act on. Then they define cases, and use the **goto** keyword to indicate which state to transition on. If the user does not want to specify all cases, they can use _ to match all values.

Switch Case :

```
switch (<input_name>,...) {
      case <value>, ... : goto <state_dest>
}
```

The second way to specify transitions is an if-statement. This works with both with transitions on inputs and transitions and any pre-defined variables. Within the conditional of the if-statement, the user can check for transition values. Using the **goto** keyword the user signifies which state they wish to transition on within the curly braces.

If:

```
if <condition> {
      goto <state>
}
```

Main Function Declaration:

This is where the user simulates the FSM! There are several pre-defined functions they can perform on their FSM such as checking for reachability and simulating on a certain input. Or, they can define a new action to perform on the FSM. Within fill() the user must first access their FSM. This is done the same way as any variable declaration:

```
void fill() {
      <fsm_name> <fsm_instance_name>
      /~ code block ~/
      return <return_value>
}
```

If the user wants to access certain values, such as the output, from within an fsm, they use the '.' operator. The output is a list of all the outputs corresponding to each state output. If a cycle is not specified then the output is the list of all outputs from every cycle.

```
 out = <fsm_instance_name>.output[<cycle_number>] ~cycle optional
```

Cool Things:

There are some interesting things that one can do with Sake. The user can create an array of identical FSMs within the FSM declaration:

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

This is useful for defining identical, concurrent FSMs as done in the binary counter code example.

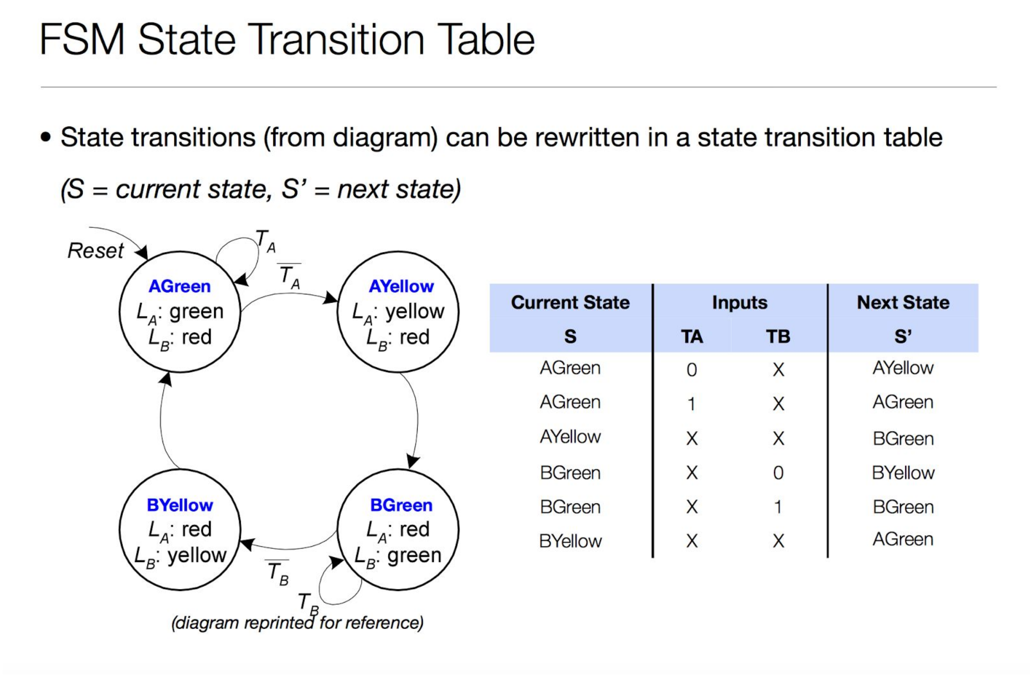
The user is also able to define concurrent FSMs. This is done within the transition code. After the **switch** keyword, the user defines not only the input on which they wish to transition, but also an fsm that they defined previously. The case will then decide based on the transition value and on the other FSM's state:

```
switch(<input_name>, <fsm_name>) {
      case <value>, <concurrent_state_name> : goto <state_name>
}
```

The if-statement is even simpler. Within the conditional, the user references any FSM and state in order to decide whether to transition.

```
<fsm_name> <fsm_instance_name>
if <fsm_instance_name>.<state> && ... {
           goto <state>
}
```

Sample code:



FSM State Transition Table

- State transitions (from diagram) can be rewritten in a state transition table

  (S = current state, S' = next state)

(diagram reprinted for reference)

Current State	Inputs		Next State
S	**TA**	**TB**	**S'**
AGreen	0	X	AYellow
AGreen	1	X	AGreen
AYellow	X	X	BGreen
BGreen	X	0	BYellow
BGreen	X	1	BGreen
BYellow	X	X	AGreen

For our sample code we will be describing a Moore machine for traffic lights at a crossing (shown above). If there is no traffic on route A, we transition to AYellow and on the next clock tick, to BGreen, and vice-versa for route B.

However, we will be adding concurrency to it. We will first describe an FSM to simulate traffic for routes A and B given the number of cars on that route (FSM traffic). Then we will describe the traffic light FSM that will depend on states in the traffic FSM in addition to TA and TB.

If there is heavy traffic on a certain route, but light or medium traffic on the other, we open up the heavy traffic route despite there being cars on the light route. Traffic[0] models route A and Traffic[1] does route B.

For the second example, we simulate a 32-bit binary counter using 32 1-bit FSMs that depend on each other. We work on both of these CSMs in our main function, “fill()”

Traffic

```
fsm traffic[2]{
    input [int cars]
    output [string load]

    state None | Light | Medium | Heavy

    start None (“None”) {
        if cars>0 {goto Light}
        else {goto None}
    }
    Light (“Light”) {
        if cars=0 {goto None}
        if cars>10 {goto Medium}
        else {goto Light}
    }
    Medium (“Medium”) {
        if cars<=10 {goto Light}
        if cars>50 {goto Heavy}
        else {goto Medium}
    }
    Heavy (“Heavy”) {
        if cars<=50 {goto Medium}
        else {goto Heavy}
    }
}
```

Traffic_light

```
type bit = 0 | 1
type light = “red” | “yellow” | “green”

fsm traffic_light {
    input [bit ta, bit tb]
    output [light la, light lb]
    state AGreen | AYellow | BGreen | BYellow

    start AGreen (“green”,”red”) {
        switch(ta,traffic[0],traffic[1]) {
            case _,Light,Heavy : goto AYellow
            case _,Medium,Heavy : goto AYellow
            case 1,_,_ : goto AGreen
            case 0,_,_ : goto AYellow
        }
    }

    AYellow (”yellow”,”red”) {
        goto BGreen
    }

    BGreen (”red”,”green”) {
        switch(tb,traffic[0],traffic[1]) {
            case _,Heavy,Light : goto BYellow
            case _,Heavy,Medium : goto BYellow
            case 1,_,_ : goto BGreen
            case 0,_,_ : goto BYellow
        }
    }
    BYellow ("red","yellow") {
        goto AGreen
    }
}
```

Binary Counter

```
type bit = 0 | 1 ~ limit bits to 0 or 1

fsm  bincount[i] {
    input []
    output [bit o]
    state = On | Off

    tgl = 1

    for j in (0:i){
        tgl = tgl && output(bct[j])
    }

    start Off (0){
        if tgl=1 {goto On}
        if _ {goto Off}
    }
    On (1){
        if tgl=1 {goto Off}
        if _ {goto On}
    }
}
```

Sake code for Traffic light and Binary counter

```
void fill() {
    type bit = 0 | 1
    traffic_light f
    traffic t
    int carNumber
    bit ta
    bit tb
    while (carNumber,ta,tb = sysin) {
        f.sim(ta,tb,t.sim(carNumber))
        if f.output[0] == “red” {
            print(“Halt on A! In the name of the law!”)
        }
        elif f.output[0] == “yellow” {
            print(“Slow down on A!”)
        }
        else {
            print(“You’re free to go on A!”)
        }
    bincount b[32]
    b.tick(5)
    print(b.output[4])
    f.reset()
    f.is_reachable(AGreen, 4)
}
```