

# FSM Workbench - Interim Report

Stephen Jeapes

Monday 26<sup>th</sup> January 2004

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project Objectives . . . . .	1
1.2	Why ML? . . . . .	2
<b>2</b>	<b>Progress</b>	<b>2</b>
2.1	Input . . . . .	3
2.1.1	File Specification . . . . .	3
2.1.2	Generation of the Lexer . . . . .	3
2.2	Parser and Simulator . . . . .	4
<b>3</b>	<b>Project Plan</b>	<b>4</b>
3.1	Core Objectives . . . . .	4
3.2	Milestones . . . . .	6
3.3	Optional Objectives . . . . .	6
<b>4</b>	<b>Project Timing</b>	<b>7</b>
4.1	Milestones . . . . .	7
4.2	Individual Task Timing . . . . .	7
<b>5</b>	<b>Risks and Fallbacks</b>	<b>8</b>
5.1	Graphical System . . . . .	8
5.2	Other Risks . . . . .	8

## 1 Introduction

### 1.1 Project Objectives

The objective of the project is to produce a toolbox of functions that can be used on their own or as part of a larger Matlab style program. General sets of functions to be performed on FSMs should be implemented but the functionality of the toolbox should include functions required to compose Protocol Transducers as detailed in [6]. The toolbox is to be written in the

ML programming language (see section 1.2 for more details). The functions within the toolbox should be as flexible as possible and be able to handle several different types of FSMs.

## 1.2 Why ML?

Much detail can be found in chapter 1 of [1].

ML (or more specifically the Standard ML or New Jersey (SML/NJ) variant of the ML language) is a functional language. As such the computation occurs by definition of functions rather than by control flow statements. As ML is a functional language there is *freedom from side effects*. This means that unlike in a conventional language where  $a = b + c$  would produce a new (or changed) value for  $a$ , in ML this would create an entirely new value (even in the case where the variable,  $a$ , had been given a value earlier on in execution).

As mentioned above computation occurs via definitions and application of functions rather than via control-flow statements. As this is the case ML encourages recursion rather than for-else statements etc. ML allows a very elegant style of recursion and this is supported by the easy use of rule-based programming.

Higher-Order functions are also an important part of ML as it allows a function to be passed as an argument and a generalized version of the function is returned.

Polymorphism is another important feature of ML. Where in other languages one would have to use function overloading to define functions to deal with several different types of arguments it is possible in ML to define a single function that can handle arguments of many different types.

ML is a strongly typed language which means often errors produced in the debugging process are easier to spot. Unlike many other strongly typed languages the ML types system infers all types of variables directly from the code with no need to define the type of arguments of function when they are defined (although this too is possible)

ML also has a powerful module system that can hide functions from the 'outside world' whilst allowing other modules and code to access a defined set of functions.

Further details of the ML language can be found in [1] and [2].

## 2 Progress

Currently there are three sml files to the project, `fsm_blif.lex`, `readin.sml`, `fsm_parse.sml` and `simulate.sml`, the files read in data from a file, convert the data from the file into a useful form and place it in a data structure and simulate the fsm respectively.

## 2.1 Input

### 2.1.1 File Specification

The first task of the program is to take a specification of a Finite State Machine in a file and read in the information before it can be processed. It is possible to represent FSMs using the Berkeley Logic Interchange Format (BLIF) [3]. For the purposes of this project I will be using a modified version of the Berkeley Logic Interchange format. The format is shown below with descriptions of the terms and differences to the the BLIF.

```
.startkiss
.i <num-inputs>
.o <num-outputs>
.p <num-terms>
.s <num-states>
.r <reset-state>
<input> <current-state> <next-state> <output>

:

<input> <current-state> <next-state> <output>
.end_kiss
```

The main differences between this specification is that the last three parameters before the list of transitions are required rather than being optional (as the BLIF Specifies). The state names in both  $i_j$  and  $j_i$  must be formed from alphabetical characters and then can be optionally followed by several digits. Other text can exist before `.startkiss` and after `.endkiss` but this will be ignored by the lexer and parser.

### 2.1.2 Generation of the Lexer

The file `fsm_blif.lex` contains the information required by ML-Lex to generate a lexical analyzer for the input file.

ML-Lex takes a file of the form

```
user declarations %% ML-Lex definitions %% rules
```

The user declarations include the data-type `lexresult` which is the type that the lexer returns. The rules are composed of a set of regular expressions and actions to perform, an example is shown below.

```
{alpha}+{digit}* => (state yytext);
```

In the example above both alpha and digit have been defined in the ML-Lex definitions section. The example line above looks for one or more letters followed by zero or more numbers and outputs a token of the type lex result.

Given the complete definitions file for generating the lexer the command `m1-lex <filename.lex>` generates the file `<filename.lex.sml>` which contains the sml code to return a lexer token from an input stream. The file `readin.sml` uses the code generated by ML-Lex to produce a list of tokens from an input file.

The full documentation for ML-Lex is available online [4]

## 2.2 Parser and Simulator

By loading `simulate.sml` (and both `readin.sml` and `fsm_parse.sml` from statements in the file) into the sml environment it is then possible to call the function `next_state` passing it the input, the current state and a list of transitions, the function will then return the output and the next state.

The parser is a function that parses the header and the list of transitions and returns a record representing the FSM. The individual functions to parse the head and the body of the data file extract the appropriate data in the required type from tokens.

## 3 Project Plan

The following sections outline the tasks and time-scales for the development of the project. The optional objectives detailed in sections 3.3 would be suitable extension of the project if the core objective are satisfactorily completed before time.

### 3.1 Core Objectives

1. Graphical interface to the functions

This would be implemented using the eXene libraries provided in the SML/NJ distribution. These libraries allow the programmer to produce standard graphical interfaces for the X-Windows system used by the Unix/Linux operating systems. The basic interface (excluding the drawing of a state diagram, see point 3 below) should not be very challenging from a interface design point of view but it is important to become familiar with the eXene libraries and the methods required to build a GUI.

2. Input of FSM into an appropriate Data structure from a text file

Whilst the current code performs the reading in of an FSM representation and placing it into a data structure the data structures will need

adjusting to handle product FSMs (and to allow timed FSMs to be represented should time allow).

3. Graphic Representation of FSM

This requires a state diagram to be drawn from the representation of the FSM. To do this and produce a useful output is by no means a trivial task. There exist several methods of flattening graphical structures and these will need investigating before an algorithm can be implemented.

4. Simulation of FSM

This point requires the inputs to the FSM to be read in from a specified text file and a number of cycles to be simulated. The main challenge in the objective is to provide an easy to use interface that will be usable for a large project rather than trivial examples.

5. Ability to handle normal FSMs and Product FSMs

This objective is encapsulated in point 2.

6. Composition of Product FSM from 2 FSMs

7. Knock out of dead states, propagation of knocked out states to generate control FSM

This step is required to knock out dead end states once a product FSM has been composed.

8. Determinise FSM

This is to determinism the outputs of an FSM such that the data transfer is as fast as possible.

9. Find the shortest path from one state to another

This optimizes the latency of the FSM.

10. Identification and Removal of redundant states

This will require several methods of removal of redundant states to be evaluated and the most suitable one chosen for the best coding style in SML.

11. Ability to change timed FSMs into normal FSMs with a linear series of states

Rather than implement timed FSMs directly a simple way to represent timed FSMs is as a linear series of states, this objective requires the input of data to be modified so that the timings can be read in and then converted to a list of transitions as the input file is parsed.

12. Output results of simulation and product FSM produced

This objective requires an output format to be decided and implemented

### 3.2 Milestones

The milestones below bring together the core objectives together in such a way that each milestone will represent a fully functional unit, the list is ordered in chronological order of expected completion.

1. Ability to handle normal and product FSMs and to compose the latter from two FSMs
2. Graphical interface without display of state diagram
3. Suitable output of simulation results and composed product FSMs
4. Can perform operations described above on FSMs (e.g. knocking out of states)
5. Graphical interface with display of state diagrams

Item two is effectively a graphical interface to the current code. Expected completion dates for these milestones are in section 4.1.

### 3.3 Optional Objectives

The following items are considered to be merely desirable and hence will not form a part of the core project. These features may be implemented if the core objectives are completed satisfactorily.

- Editing of FSM  
The core objectives do not allow the FSM to be edited once it has been read in from the input file. Ideally it would be preferable to be able to edit the FSM (even using a dialog to edit the transition rather than by moving objects on the state diagram). This would also require the modified FSM to be written back to a text file to record the changes
- Graphical output of FSM Simulation  
The core objectives merely require the simulation output to be text and recorded in a file after simulation, for clearer understanding of the operation of a complex FSM a graphical output would be preferable.
- Ability to handle Timed FSMs  
Point 11 allows the project to handle timed FSMs but using a rather primitive method, direct implementation of timed FSMs would be preferable.

## 4 Project Timing

### 4.1 Milestones

The milestones listed above should be reached by the following dates

1. Ability to handle normal and product FSMs and to compose the latter from two FSMs  
7<sup>th</sup> March
2. Graphical interface without display of state diagram  
29<sup>th</sup> May
3. Suitable output of simulation results and composed product FSMs  
4<sup>th</sup> June
4. Can perform operations described above on FSMs (e.g. knocking out of states)  
25<sup>th</sup> March
5. Graphical interface with display of state diagrams  
14<sup>th</sup> June

### 4.2 Individual Task Timing

Below is a table that shows each individual task required and the expected duration to reach each milestone. Optional objectives are not included. The timings below show the times all time were to be dedicated to each task, the gantt chart below shows the actual timings with time shared between several resources.

1. Ability to handle normal and product FSMs and to compose the latter from two FSMs  
Altering of data-type to suit both product and normal FSMs - 1 day  
Research of composition of product FSMs - 2 day  
Design of algorithm to compose FSM - 5 days  
Implementation of algorithm - 4 days
2. Graphical interface without display of state diagram  
Research into Graphical systems for SML - 7 days  
Research into chosen Graphics libraries - 5 days  
Design of interface - 2 days  
Coding - 6 days
3. Suitable output of simulation results and composed product FSMs  
Design of output format and control of system - 3 days

Coding - 3 days

4. Can perform operations described above on FSMs (e.g. knocking out of states)

Design of modules and function to be performed - 6 days

Design of individual functions - 6 days

Coding - 6 days

5. Graphical interface with display of state diagrams

Research into graphical structure flattening - 6 days

Design of algorithm - 10 days

Coding - 10 days

## 5 Risks and Fallbacks

### 5.1 Graphical System

Currently there are two possible systems for implementing graphical user interfaces for SML code. eXene is the system currently being integrated into the SML basis library however there also exists another system called SML.tk. As neither SML.tk or eXene are fully mature there are risks with using either system. In the research of both systems it will be important to concentrate on the stability along with the systems features and ease of use. The existence of two systems gives a natural fallback if one systems turns out to be inadequate.

### 5.2 Other Risks

Other risks include the possibility of including too many features into the system for the time available. This can be avoided by designing and coding the functions in stages and limiting the length of this stage of the project.

## References

- [1] Ullman. J, Elements of ML Programming, ML97 Edition, *Prentice Hall*, New Jersey USA, 1998.
- [2] [www.smlnj.org](http://www.smlnj.org)
- [3] Arno Wagner, 4th December 1998  
Finite State Machine Descriptions - Berkeley Logic Interchange

<http://www.bdd-portal.org/docu/blif/node8.html>  
(Accessed 7th October 2003)

- [4] Andrew W. Appel, James S. Mattson, David R. Tarditi  
*A lexical analyzer generator for Standard ML*  
Version 1.6.0, October 1994  
<http://www.smlnj.org/doc/ML-Lex/manual.html>  
(Accessed 20th December 2003)
- [5] Reppy, John H, Gansner, Emden R.  
*The eXene Library Manual*  
Version 0.4, February 11, 1993,  
<http://www.smlnj.org/doc/eXene/>  
(Accessed 12th January 2004)
- [6] Clarke T. J. W., Androutsopoulos V.  
*Protocol Converter Synthesis*  
Imperial College London