Soon: A Regular Description Language for Simulating Finite Automata

Reese Bunker
College of Computing
Michigan Technological University
Houghton, MI USA
rjbunker @mtu.edu

ABSTRACT

Soon is a description language for Finite Automata, as well as a simulator allowing a list of test inputs. Many simulators for automata exist, but most opt to use a GUI to describe automata. The syntax of Soon allows for a quick description of finite automata. The regularity of Soon's syntax allows for the creation of a self-accepting description of a finite automata.

CCS CONCEPTS

• Formalisms • Theory of computation • Regular languages • Formal languages and automata theory

KEYWORDS

Description Language, Finite Automata, Simulator

Reference format:

Reese Bunker. 2023. Soon: A Regular Description Language for Simulating Finite Automata. Infinite Loop, volume 1. *Michigan Technological University, Houghton, MI, USA*

Introduction

Soon is a description language for Finite Automata, as well as a simulator allowing a list of test inputs. Users input a Soon description file and a string input file, and Soon will run the input file through the automata described in the Soon description file, outputting either 0 or 1. 1 meaning the input file is accepted through the automata, 0 meaning it is not accepted. Soon was written in C using GNU's Bison and Flex [3]. Much of the Flex/Bison boilerplate code came from Introduction to Compilers and Language Design [2].

Motivation

While many programs exist for simulating finite automata, most require users to draw automata using a GUI [4, 6]. Although intuitive to use, it is time-consuming for larger automata. Having a typed input for automata allows for quicker transfer of handwritten automata to a simulator. Notably, Dr. David Doty's Automaton simulator uses a similar syntax [5]. The regularity of the language is an interesting feature, as traditional parsing techniques such as top-down and bottom-up may be unnecessary. It is also possible for Soon descriptions to act as a file format to be generated and interpreted.

Syntax of Soon

Although a Soon description is a definition of a finite automaton. The 5 statements in a Soon description define an automaton in a way that corresponds to the formal definition of an automaton [1, pg 35]. Listed below is the general structure of a Soon description.

```
states: <list of states> ;
alphabet: <list of characters> ;
start: <start state> ;
accepting: <list of accepting states> ;
transitions: <list of transitions> ;
```

The states declaration provides Soon with a list of states in the automata. This allows the interpreter to allocate space for the transition table. The list of states should contain only integers between 0 and 255. For example, if a automata contains 3 states, the states declaration may read like this:

```
states: {0, 1, 2};
```

However, the states do not have to in order, nor do the integers have to be adjacent. The following states declaration is also valid:

```
states: {255, 13, 5, 0};
```

It is possible to imply all numbers in between two numbers inclusively using a hyphen:

```
states: {0-12};
```

The declaration of the alphabet provides Soon with the alphabet for the given automata. All ASCII characters with values between 33 and 126 inclusively are valid characters. For an automata with an alphabet containing letters a, b, c and d, the alphabet declaration could be written as:

```
alphabet: {'a', 'b', 'c', 'd'};
```

Alternatively, if the alphabet contains characters that appear consecutively on an ascii table, the declaration could also be written as:

```
alphabet: {'a'-'d'};
```

For more complex alphabets, both of these methods can be used interchangeably.

```
alphabet: {'a'-'d','f','0'-'9',''',';' };
```

The declaration of the start state informs the interpreter at which state to begin the simulation at. The start state should be a member of the states set. For example, if the list of states is {0, 1, 2}, and 0 was to be the start state, the start declaration would be written as:

```
start: 0;
```

The definition of accepting states allows Soon to know if a given string is accepted or rejected when the automaton finishes reading the input. The list of accepting states should contain only integers between 0 and 255 that are members of the list of states. For example, if there is a list of states {0, 1, 2} and state 0 is accepting, then the definition of the accepting state should be written as:

```
accepting: {0};
```

There may be multiple accepting states. For example, if there is a list of states {0, 1, 2} and states 0 and 2 are accepting, then the accepting states definition could be written as:

```
accepting: {0, 2};
```

Likewise, with the states definition, the order of the elements in the list of accepting states is unimportant. Presently, unlike the states definition, the accepting states must be listed one by one.

The final statement required for a Soon description is the transition statement. Each member of the list should be a 3-tuple with members (integer, char, integer). Each member of the tuple corresponds to: the current state, input character, and next state. For example, the 3-tuple (0, 'a', 1) corresponds to a transition from 0 to 1 on 'a'. Each 3-tuple should be separated by commas. Given a simple automaton with two states {0, 1} and 1 transition from 1 to 0 on 'b', and a transition from 0 to 1 on 'a'. The transition statement would look like:

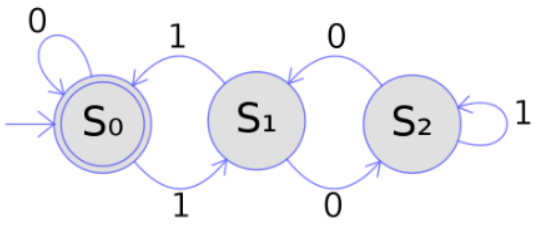
```
Transitions: {(0, 'a', 1), (1, 'b', 0)};
```

A transition shared by many inputs may be written using the following shorthand:

```
Transitions: {(0, 'a'-'c', 0)};
```

Example Soon Description

Using the syntax above it is relatively straightforward to generate a Soon description given an image of an automaton. For example given the following automaton:



- Sourced from Wikimedia commons[7]
- Automaton accepting binary strings representing value n such that n ≡ 0 (mod 3)

The Soon description can be written as such:

```
states: {0, 1, 2};
alphabet: {'0', '1'};
start: 0;
accepting: {0};
transitions: {(0, '0', 0), (0, '1', 1), (1,
'0', 2), (1, '1', 0), (2, '0', 1), (2, '1',
2)};
```

Creating regular expressions for Soon

To evaluate the regularity of the Soon syntax, the following definition of a regular language will be used:

Definition 1.16: A language is called a regular language if some finite automaton recognizes it [1, p.40].

Essentially, if an automaton can be constructed that recognizes the underlying language of a Soon description. Then that language is regular. To simplify the problem, each statement in a Soon description will be denoted as its own language. The following notation will be used to represent these languages.

Q = the language describing the set of states on an automaton
α = the language describing the alphabet over an automaton
S = the language describing the start state of an automaton
A = the language describing the set of accepting states in an automaton
T = the language describing the transition function of an automaton

Q◦α◦S◦A◦T will represent the language of a Soon description. By theorem 1.26, if all 5 languages are regular, then the concatenation of these languages will be regular.

Theorem 1.26: For any regular languages A1 and A2, A1◦A2 is regular [1, pg 47].

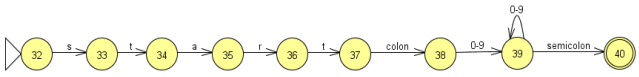
As finite automata are equivalent to regular expressions [1, pg 66], regular expressions can be used to define behavior for automata. To create these expressions, some special notation must be defined. Parentheses are reserved characters, thus square brackets will be used in their place.
For example:

a[a, b]*b will denote the all strings starting with a and ending with b. The character μ will be used to represent any character with an ascii value between 33 and 126 inclusively. D will be used to denote any digit 0-9. A plus sign may be used to represent at least one concatenation [1, pg 65]. The regular expressions for each are as follows.

Q = states: { [D+ ∪ D+-D+] [,D+ ∪ ,D+-D+]* };
α = alphabet: { ['μ' ∪ 'μ'-'μ'] [,'μ' ∪ ,'μ'-'μ']* };
S = start: D+ ;
A = accepting: { D+ [, D+]* };
T = transitions: { [(D+ , ['μ' ∪ 'μ'-'μ'] , D+)] [, (D+ , ['μ' ∪ 'μ'-'μ'] , D+)]* };

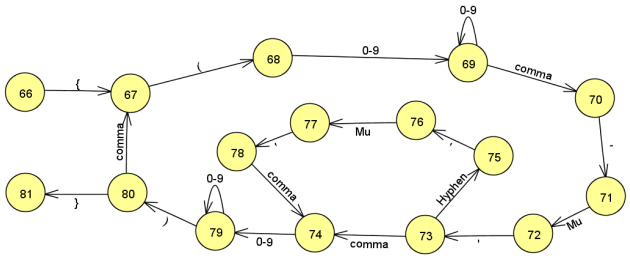
Creating Automata to Accept Soon Descriptions

With the above regular expressions, it is possible to construct automata to accept these languages. For example, listed is a finite automaton accepting language S.



- Image generated using JFLAP 7.1 [4].
- Automaton representing the language S

A more complex automaton is that of the language T. Listed below is an abbreviated look at the finite automaton for accepting language T.



- Image generated using JFLAP 7.1 [4].
- Automaton representing part of the language T

Since Q, α, S, A and T, can be represented using finite automata, they can all be considered regular. Therefore Q◦α◦S◦A◦T is regular by Theorem 1.26 [1, pg 47]. As Q◦α◦S◦A◦T is regular, an automaton to accept all syntactically correct Soon descriptions may be created. Listed below is the full Soon description for an automaton to accept Q◦α◦S◦A◦T.

```
states: {0-82};
alphabet: {'!'-'~'};
start: 0;
accepting: {82};
transitions: {
(0, 's', 1), (1, 't', 2), (2, 'a', 3), (3,
't', 4), (4, 'e', 5), (5, 's', 6), (6, ':',
7), (7, '{', 8), (8, '0'-'9', 9), (9, '0'-
'9', 9),

(9, ',', 8), (9, '-', 10), (9, '}', 12),
(10, '0'-'9' , 11), (11, '0'-'9', 11), (11,
',', 8), (11, '}', 12), (12, ';', 13),

(13,'a', 14), (14, 'l', 15), (15, 'p', 16),
(16, 'h', 17), (17, 'a', 18), (18, 'b',
19), (19, 'e', 20), (20, 't', 21),
(21, ':', 22), (22, '{', 23), (23, ''',
24), (24, '!'-'~', 25), (25, ''', 26), (26,
',', 23), (26, '-', 27), (27, ''', 28),
(28, '!'-'~', 29), (29, ''', 30), (30, '}',
31), (30, ',', 23), (26, '}', 31), (31,
';', 32),

(32, 's', 33), (33, 't', 34), (34, 'a',
35), (35, 'r', 36), (36, 't', 37), (37,
':', 38), (38, '0'-'9', 39), (39, '0'-'9',
39),
(39, ';', 40),

(40, 'a', 41), (41, 'c', 42), (42, 'c',
43), (43, 'e', 44), (44, 'p', 45), (45,
't', 46), (46, 'i', 47), (47, 'n', 48),
(48, 'g', 49), (49, ':', 50), (50, '{',
51), (51, '0'-'9', 52), (52, ',', 51), (52,
'0'-'9', 52), (52, '}', 53), (53, ';', 54),

(54, 't', 55), (55, 'r', 56), (56, 'a',
57), (57, 'n', 58), (58, 's', 59), (59,
'i', 60), (60, 't', 61), (61, 'i', 62),
(62, 'o', 63), (63, 'n', 64), (64, 's',
65), (65, ':', 66), (66, '{', 67), (67,
'(', 68), (68, '0'-'9', 69), (69, '0'-'9',
69), (69, ',', 70), (70, ''', 71), (71,
'!'-'~', 72), (72, ''', 73), (73, ',', 74),
(73, '-', 75), (75, ''', 76), (76, '!'-'~',
77),
(77, ''', 78), (78, ',', 74), (74, '0'-'9',
79), (79, '0'-'9', 79), (79, ')', 80), (80,
',', 67), (80, '}', 81), (81, ';', 82)
};
```

Conclusion

Although the language Soon is regular, many strings in its language may not be semantically valid. For example, to create a description with transitions to states that are not in the states set. Nondeterministic finite automata may also be described using Soon. However, the Soon interpreter does not have functionality to handle nondeterminism. While the syntax and structure of Soon are simple, larger definitions of automata can become difficult to read and use. Soon may be better suited as an intermediate representation, to be generated by regular expressions, or by concatenation of existing automata. Future work for this project may include support for nondeterminism and conversion to deterministic automata.

REFERENCES

[1] M. Sipser, Introduction to the Theory of Computation. Boston, MA: Cengage Learning, 2013.
[2] D. Thain, Introduction to Compilers and Language Design. Douglas Thain, 2023.

[3] “GNU Bison - the yacc-compatible parser generator - GNU Project - Free Software Foundation,” gnu.org, https://www.gnu.org/software/bison/manual/ (accessed Oct. 30, 2023).
[4] “JFLAP Version 7.1,” JFLAP, https://www.jflap.org/ (accessed Oct. 30, 2023).
[5] D. Doty, Automaton Simulator, https://web.cs.ucdavis.edu/~doty/automata/ (accessed Nov. 1, 2023).
[6] G. Silber, FSM Simulator, https://www.eecis.udel.edu/~silber/automata/ (accessed Nov. 1, 2023).
[7] User:Mikm, DFA Example Multiplies of 3. Svg. Wikimedia, 2007.