Documentation for wsn2fsm

Frank Kusters

June 8, 2008

Contents

1 NAME

wsn2fsm: Wirth syntax notation to finite state machine compiler

2 SYNOPSIS

```
wsn2fsm.pl inputfile
```

3 DESCRIPTION

`wsn2fsm.pl` will take a file as its argument. The file must contain a language defined in the Wirth syntax notation. It will then produce a finite state machine which accepts the same language and prints it to STDOUT.

4 GLOBAL VARIABLES

4.1 $debug

If `$debug` is set to `true`, the script will print debug information to STDOUT.

4.2 $current_token

A string containing the last token read by `next_token()`.

4.3 %rules

A hash containing all rules. The rule names are used as keys, and the values are references to hashes (explained in "Data Structures").

4.4 $nodeid

The id that a newly created node will get in the generator. This number is incremented by `new_node()`.

4.5 @results

An array of strings containing the output strings of the generator. Each string represents one node, in the following format: id start_node end_node transition probability

4.6 $epsilon

A data structure representing the epsilon. This data structure must not be changed at runtime.

4.7 $main

A string specifying the rule name of the starting (main) rule.

4.8 %stack

Keeps the current list of rules that are being parsed. If a rule would appear twice in the list, this would automatically mean that there is some kind of self-referencing going on, which currently cannot be handled by the application.

4.9 @readtokens

An array containing the tokens that are read from the current line of the input file.

5 FUNCTIONS

5.1 main()

Parameters: none. Returns nothing.

This function starts the tokenizer, parser, preprocessor and generator (essentially reading the file, processing it and creating the output).

5.2 generate()

Parameters:

a reference containing the rule to generate output for

an integer with the id of the start node

an integer with the id of the end node

a number with the probability of the rule

Returns nothing.

Generates the finite state machine output for the rule specified.

5.3 new_node()

Parameters: none.

Returns the new value of `$nodeid`.

Increments `$nodeid` and returns its new value.

5.4 print()

Parameters: none. Returns nothing.

Prints the array of strings representations of the nodes of the finite state machine. Each string is prepended with a unique id.

5.5 preprocess()

Parameters: none. Returns nothing.

Processes the data structure generated by the parser for use by the generator. First it disables all rules surrounded with <<>>. Then all identifiers inside the rules are replaced by the rules they identify, leaving the main rule without any identifiers inside. Finally all rules except the main rule are deleted.

5.6 disable_rules()

Parameters: none. Returns nothing.

It modifies the %rules data structure such that any identifier starting with surrounded with <<>> is transformed to the same identifier, but with only <>. If the new identifier already exists, the application will quit with an error message. The rule is also replaced by the empty rule.

5.7 replace_vars()

Parameters:

a reference containing the rule in which to replace the identifiers by their rules.

Returns nothing.

Replaces all identifiers inside the rule by the rules they identify, leaving the rule without any identifiers inside.

5.8 parse_rule()

Parameters: none. Returns nothing.

Parses one rule of a WSN by calling the different methods that parse parts of a rule in order. The rule is retrieved from the file specified on the command line and then stored in %rules.

5.9 parse_definition()

Parameters:

a reference containing the rule to which the new part of a definition must be added.

Returns nothing.

This method parses a new part of a rule definition. It is recursive; extra parts of a definition are parsed by itself.

5.10 level_down()

Parameters:

a string containing the current token.

a reference containing the rule to which to add the current token.

a string containing the type of this part of the rule.

Returns nothing.

Creates a new subrule with $current_token in the right part. It either replaces $ruleref if that is empty, or stores $ruleref in the left part of the new subrule if $ruleref is not empty.

5.11 `level_up()`

Parameters:

a reference containing the rule to which to add the current token.

a string containing the type of this part of the rule.

Returns nothing.
Creates a new subrule and stores `$ruleref` in the left part of the new subrule.

5.12 `check_structure()`

Parameters:

a reference containing the subrule to check for validity.

Returns nothing.
Checks the specified subrule for validity of the data. This check is only for programming errors. If it finds errors that means the parser has created a wrong data structure. When an error is found, a message is displayed. In case the type is not defined the program immediately exits, because it can't handle this in the generator.

5.13 `processor_error_programmer()`

Parameters:

a boolean which specifies whether the application exits or not after printing the error message.

Returns nothing.
Prints an error message about the incorrect data structure to STDERR. It then exits the application if `$fatal` is `true`.

5.14 `skip_whitespace()`

Parameters: none. Returns nothing.
Advances the input token to the first element that is not whitespace.

5.15 `parse_rulename()`

Parameters: none. Returns nothing.
Identifies whether the current token is the name of a rule. That is the case if it starts with a dollar sign ('$'). If another rule was already defined with the same name, the program exits with an error message specifying the problem. The new rule is added to the `%rules` data structure.

5.16 `parse_assignment()`

Parameters: none. Returns nothing.
Identifies whether the current token is an assignment (':=='). Exits the application with an error message if it's not.

5.17 `parse_end_of_rule()`

Parameters: none. Returns nothing.

Identifies whether the current token is the end of a rule (';'). Exits the application with an error message if it's not.

5.18 `is_whitespace()`

Parameters: none. Returns `true` if the current token is whitespace, `false` if it's not.

Determines whether the current token is whitespace (space, tab or newline). Returns `true` if the current token is whitespace, `false` if it's not.

5.19 `log_condition()`

Parameters:

a string containing the condition to print.

Returns nothing.

Prints `$condition` if the global variable `$debug` is true.

5.20 `error()`

Parameters:

a string containing the expected token.

Returns nothing.

Prints an error message about an unexpected token and exits the application. The error message also contains the line number and file name of the file being parsed. Uses the `$current_token` global variable.

5.21 `next_token()`

Parameters: none.

Returns the token read or undefined when EOF is reached.

Reads the input file (specified on the command line). Divides it into tokens by several delimiters and returns token by token. The token returned is also stored in `$current_token`. When the end of the file is reached, returns undefined and makes `$current_token` undefined. If `$debug` is `true`, prints the token returned.

6 REMARKS

To change the output file format, edit the `generate()` function. A marker is placed at the lines to edit. More information is also found there.

7 DATA STRUCTURES

There is one major data structure used throughout the application. It is a hash (`%rules`) representing all rules. The keys of the hash are the rule identifiers. The values are references to binary trees. They are references to be able to replace the hashes altogether in `level_up` and `level_down`. The nodes and leafs of the binary trees are expressed as hashes. The keys of those hashes are:

type

This string specifies the type of data to be found in the hash. If it is 'NIL', then the key 'value' must exist, making the current hash a leaf. If it is any other value, the keys 'left' and 'right' must exist, making the current hash a node. Other possible values are: 'OR', 'AND' and 'AND_ONCE', where `left` is 'OR'-ed or 'AND'-ed with `right`, and 'AND_ZERO_OR_MORE' and 'AND_OPTIONAL', where `left` is 'AND'-ed with `right`, but right is treated specially, according to the value ('optional' is zero or one).

value

The string containing the text literal. This string can be surrounded by double quotes. If that is not the case, the double quotes are added in the generator. There are two special values for this element: "___EMPTY___" and "___EPSILON___", both without the quotes. "___EMPTY___" is used for disabled rules (double angle brackets: "<<>>"). In the FSM, it means that the state transition can be made, no matter what the input. "___EPSILON___" is used for optionals, zero or more, etc. A state transition can be made when reading a zero-length string.

Both special values can be used in the input files, and they will be treated as such (which is especially of value for "___EPSILON___"). To use them as string literals, surround by quotes.

left

Since every element is part of a binary tree, this is a reference to a new leaf or node. `left` only contains a leaf or a node if `right` also contains a leaf or node. See `type` for more info.

right

`right` is also a reference to a new leaf or node. For more info see `left` and `type`.

probability

`probability` is a floating point value higher than 0.0 and lower than or equal to 1.0. It only has significance when used in a node or leaf below an 'OR'-node. If it is used in any other situation, the resulting probability will be treated is if it was below an 'OR'-node. In the resulting FSM, the logarithm of the probability is shown for the corresponding transition.

8 AUTHOR

Frank Kusters

9 COPYRIGHT