HSM Compiler User Guide

Stephen Waits

June 24, 2004

Contents

Listings

Chapter 1

Introduction

1.1 Overview

here's an overview of the this document.

1.2 Motivation

why create an HSM language and compiler?

1.3 Hierarchical State Machines

brief description on how hsm's work [Har87].

Chapter 2

Hierarchical State Machine Compiler

2.1 Overview

hsmc is a command line based compiler, named *hsmc.exe*. It reads an HSM source input file, compiles it, and outputs two files (*.cpp* and *.h*). Each HSM source input file must define at least one valid Machine. Only one input file can be compiled at a time; however, that file may contain multiple Machine definitions (see B.2.4).

2.2 Running the Compiler

Invoking *hsmc* from the command line should print a usage statement:

```
HSMC - Hierarchical State Machine Compiler

Usage: hsmc [-hdpq] [-o prefix] source_file

  -h            this help
  -d            include debugging information
  -p            actions declared as pure virtual
  -q            quiet mode (only output errors)
  -o prefix     specify prefix for output files [prefix.cpp,prefix.h]
  source_file   HSM source file
```

To compile your HSM file using all defaults, you need only specify your source filename. For example:

```
hsmc myfile.hsm
```

outputs two files upon successful compilation: *myfile.cpp* and *myfile.h*.

2.2.1 Compiler Options

-h Prints the usage statement. This is equivalent to running *hsmc* with no parameters.

-d Include debugging information in the generated machine. This includes considerably more calls to HSMDebug. The debug strings are also available via HSMGetLastDebugMessage. The debug strings consist of descriptions of the current state, current transitions, or diagnostic messages.

-p Actions, by default, are declared virtual member methods and defined to be empty. This flag causes these methods to be declared pure virtual and remain undefined.

-q By default, *hsmc* will output status messages during parsing, compilation, and code generation. Specifying this flag causes the output to be quieted such that only errors are emitted.

-o prefix Specifies the output filename prefix. By default this is set to the input filename after it's been stripped of its extension. So, for *input.hsm*, the default output prefix is *input*.

To generate *output.cpp* and *output.h* from the input file *input.hsm*, you should specify the prefix explicitly:

```
hsmc -o output input.hsm
```

The output prefix may include a prepended directory path as well.

2.3 Compiler Output

The *hsmc* compiler outputs a C++ header file and a C++ source file.

2.3.1 Header File

The header file declares one class for each machine defined in the HSM source input file.

Each machine class enumerates events, enumerates states, declares several methods common to all machines, and declares all actions (virtual) which should be overriden before the machine is instanced.

For example, a machine declaration of:

Machine(MyMachine,16)
{
  ...
}

generates a class declaration of:

```
class MyMachine
{
public:
   ...
};
```

Event Enumeration

A single *enum* entry is generated for each event identifier referenced in the HSM source input file. This enumeration is within the scope of the machine class.

For example, the following HSM source fragment:

```
Action(MOUSE_INPUT,BeepSound);
Transition(KEYBOARD_INPUT,NextState);
Terminate(CRITICAL_EVENT);
```

compiles to the following event enumeration:

```
// events
enum
{
  MOUSE_INPUT = 0,
  KEYBOARD_INPUT,
  CRITICAL_EVENT

  ___HSM_NUM_EVENTS___
};
```

State Enumeration

A single *enum* entry is generated for each state defined for a given machine in the HSM source input file. This enumeration is within the scope of the machine class.

For example, the following HSM source fragment:

```
Machine(MyMachine,16)
{
  State(BootUp) { ... }
  State(AwaitInput) { ... }
  State(ProcessInput) { ... }
   ...
}
```

compiles to the following state enumeration:

```
// states
enum
{
  TOPSTATE = 0,
  BootUp,
  AwaitInput,
```

```
    ProcessInput,

    ___HSM_NUM_STATES___
};
```

The logical name of the root state is *TOPSTATE*, as described in section B.4.

Common Method Declaration

These method declarations are common to every machine.

```
public:

  int  HSMGetCurrentState() const;
  bool HSMIsRunning() const;

  void HSMConstruct();
  void HSMDestruct();

  bool HSMUpdate(float dt = 0.0f);

  void HSMTrigger(int event);

  char* HSMGetLastDebugMessage();

protected:

  // debug hook (override this to trap debug messages)
  virtual void HSMDebug(char* msg);
```

int HSMGetCurrentState() const; returns an integer, the enumeration id number of the current state as enumerated in the event numeration (2.3.1). A negative integer (¡0) is returned if the machine is not running.

bool HSMIsRunning() const; returns *true* if the machine has been constructed and is presently in a valid state. returns *false* if *TOPSTATE* has exited, or the machine has not been constucted, or the machine has been destroyed.

void HSMConstruct(); Called on a non-running machine to initialize the HSM, and cause a transition into *TOPSTATE*. This does nothing if the machine is already running.

void HSMDestruct(); Called on a running machine to cause an immediate transition to *TOPSTATE*, followed by the exit of *TOPSTATE*. This does nothing if the machine is not running.

bool HSMUpdate(float dt = 0.0f); Updates any timers according to *dt*, adds an *IDLE* event to the end of the event queue, and then processes all events up to that *IDLE* event.

void HSMTrigger(int event); Adds event to the event queue. If event is invalid, it's ignored. The event queue is processed on the next call to HSMUpdate().

char* HSMGetLastDebugMessage(); Returns a private character pointer to the last debug message. This debug string is a textual description of the last operation the machine performed, such as a transition, entry into a state, or idling in a state.

See 2.2.1 for information on how to enable debugging information.

virtual void HSMDebug(char* msg); This function is empty by default and has no effect. It is called throughout the machine internals with debug messages; therefore, if you prefer immediate access to these messages, you should override this function with your own. This allows you to trap debug messages as they happen.

These debug messages are the same as are accessible via HSMGetLastDebugMessage().

See 2.2.1 for information on how to enable debugging information.

Action Method Declarations

Action methods are user defined functions which need to be called by the HSM. These actions are defined by Entry, Idle, Exit, and Action statements (see B.6).

For example, the source input fragment:

```
Entry(MyEntryFunc);
Idle(MyIdleFunc);
Exit(MyExitFunc);
Action(SomeEvent, MySomeEventFunc);
```

is compiled to:

```
protected:

  // actions
  virtual void MyEntryFunc();
  virtual void MyIdleFunc();
  virtual void MyExitFunc();
  virtual void MySomeEventFunc();
```

Note that if the pure virtual option is specified (see 2.2.1) then these methods would instead be declared as follows:

```
protected:

  // actions
  virtual void MyEntryFunc() = 0;
  virtual void MyIdleFunc() = 0;
  virtual void MyExitFunc() = 0;
  virtual void MySomeEventFunc() = 0;
```

2.3.2 Source File

The source file contains all of the inner workings of the HSM. In general, you need know nothing about this file to use *hsmc*. You may wish to browse through it in case you're curious about how the machines are implemented or if you're debugging your machine behavior.

2.4 Compiler Errors

You may encounter one or more of the following diagnostic error messages while compiling with *hsmc*. No output files are generated in the case of any error.

source line number: Invalid event queue size for Machine 'machinename' The parser was not able to determine a valid queue size in the declaration of 'machinename'. See B.4.

source line number: State 'statename' was previously defined A state named 'statename' was already found in this machine. See B.5.

source line number: multiple Default states defined in State 'statename' More than one state defined with the same name, 'statename'. See B.5.

source line number: multiple Transition's on 'eventname' defined in 'statename' More than one transition defined for a single event (i.e. ambiguous transition). See B.7.

source line number: multiple Terminate's on 'eventname' defined in 'statename' More than one terminate defined for a single event. See B.7.4.

ERROR: parser failed The parser discovered an unrecoverable syntax error. See B.2.

ERROR: unable to open 'filename' The compiler was unable to open an input file for reading or an output file for writing. Make sure the input file exists and is readable and that the output file either doesn't exist, or is writable.

ERROR: compilation aborted due to errors The compiler aborted compilation because of errors. These are most likely semantic errors and will probably have been printed in addition to this terminal diagnostic.

Chapter 3

Implementing a Machine

Step by step instructions on creating a machine, compiling it, inheriting it, and running it.

3.1 Design

3.2 Compile

3.3 Implement

Chapter 4

Examples

4.1 Traffic Signal

4.2 Soda Machine

Appendix A

Frequently Asked Questions

A.1 FAQ Category

A.1.1 Question number 1?

Answer number 1.

A.1.2 Question number 2?

Answer number 2.

Appendix B

HSM Language Reference

```
//  * A state may contain other states
//  * A state may contain Default, Entry, Idle, Exit, Transition, Action, and Terminate
//  * A single Default is allowed per state
//  * There may only be one transition per event
//  * Multiple actions are allowed per event
//  * Multiple Entry, Idle, and Exit function calls are allowed
//  * Terminate() causes all states to exit, including the root state
//  * nested states must be represented hierarchically in input file via {} pairs
//  * no empty states
```

B.1 General

The HSM language attempts to embody the most used, most important features of hierarchical state machines (as described by [Har87]). Some features from Harel's statecharts are missing from the HSM language, such as historic transitions.

The language is designed to be easy to edit and manipulate, while remaining syntactically similar to C++.

B.2 Lexical Elements

B.2.1 Character Set

The HSM source character set consists of 74 characters: the space character, the control characters representing the horizontal tab, and new-line, plus the following 71 graphical characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
{ } ( ) , ; . _ /
```

```
1  Machine(m,16)
2  {
3    // s is a default transition
4    Default(s);
5
6    // define state 's'
7    State(s)
8    {
9      // on event 'E', transition to state 'S'
10     Transition(E,S);
11   }
12
13   // define state 'S'
14   State(S)
15   {
16     // on event 'e', transition to state 's'
17     Transition(e,s);
18   }
19 }
```

Listing B.1: Case Sensitivity on Identifiers

B.2.2 Case Sensitivity

Keywords are not case sensitive. Therefore, *state*, *STATE*, *State*, and *StAtE* are all equivalent.

However, identifiers are case sensitive. For example, the HSM in listing B.1 defines two states, named S and s; as well as two events, named E and e.

B.2.3 Line Format

Whitespace is completely ignored; therefore, lines do not require separation by new-line characters, nor any special formatting with space characters.

All keywords other than *Machine* and *State* require a semicolon.

B.2.4 Source Layout

Each single HSM source input file must define at least one syntactically correct Machine (see B.4). There is no limit to the number of machines allowed in a single input file.

B.2.5 Keywords

Exactly nine keywords are defined. They are listed in listing B.2 and described in detail in sections B.4, B.5, B.6, and B.7,

Machine
State
Default
Entry
Idle
Exit
Transition
Action
Terminate

Listing B.2: Keywords

IDENTIFIER
Identifier
identifier
iDeNtifier
_identifier
_000000
_id0
id100
0id *// invalid*
100id *// invalid*

Listing B.3: Identifier Examples

B.2.6 Identifiers

Identifiers are used to name Machines, States, and Events. The first character of an identifier must be one of the 53 graphical characters in the set:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
_
```

and any characters after the first may be any of the 63 graphical characters in the set:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
_
0 1 2 3 4 5 6 7 8 9
```

Identifiers longer than 256 characters are not permitted.

Examples of valid and invalid identifiers may be found in listing B.3.

B.2.7 Numeric Literals

Numeric literals are used in the HSM language to specify event queue size (B.4) as well as time values used in timed actions (B.6.2) and timed transitions (B.7.2).

They may be whole integers or floating point numbers, and must be positive. They must be composed of the digits 0 through 9, and may include a single decimal point denoted by the period character.

B.2.8 Reserved Words

Ten reserved words are defined. They include the word *TOPSTATE* plus the nine keywords defined in B.2.5.

These words are all reserved and cannot be used as identifiers.

B.2.9 Comments

The characters // start a comment, which terminates with the next new-line character.

B.3 Scope

Scope is controlled with the curly brace characters, { and }. These apply to Machine and State definitions (see B.4 and B.5).

This scoping mechanism is what allows for the nesting of states, making these state machines hierarchical [Har87].

B.4 Machines

Machine(name,queuesize) { ... } *name* is the unique identifier for this machine; *queuesize* is the static size of the event queue and must be a whole integer greater than 0; ... is the machine definition which must qualify as a valid state definition (see B.5).

Machine is declaration and definition of the root state of a Hierarchical State Machine.

Machine is nearly synonymous with State, with the addition of requiring an event queue size specification. Therefore, everything valid in the definition of a State is also valid in the definition of a Machine. As such, a Machine definition may contain zero or more substates; however, it may not be empty. See B.5.

The state defined by Machine is the root state in the state hierarchy. It is identified by the reserved identifier *TOPSTATE* (see B.2.8), a convention set forth in [Sam02]. This automatic identification is necessary to allow explicit transitions to the root state. It is the first state entered upon machine construction.

B.5 States

May contain any number of substates.

B.6 Actions

B.6.1 Normal

B.6.2 Timed

B.6.3 Special

B.7 Transitions

B.7.1 Normal

B.7.2 Timed

B.7.3 Default

B.7.4 Termination

Appendix C

HSMC lex and yacc Source

C.1 lex, lexical scanner

```
%{

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "hsm-parser.tab.h"

int lineno = 1;

void yyerror(char* s);

%}


%%

   /* whitespace */

[ \t]+                                  ; /* skip whitespace */


   /* comments */

"//".*                                  ; /* skip C++ comments */

   /* punctuation */

","                                     { return ','; }
```

```
32  ";"                                     { return ';'; }
33  "{"                                     { return '{'; }
34  "}"                                     { return '}'; }
35  "("                                     { return '('; }
36  ")"                                     { return ')'; }
37
38
39    /* keywords */
40
41  [Mm][Aa][Cc][Hh][Ii][Nn][Ee]            { return MACHINE; }
42  [Ss][Tt][Aa][Tt][Ee]                    { return STATE; }
43  [Dd][Ee][Ff][Aa][Uu][Ll][Tt]            { return DEFAULT; }
44  [Ee][Nn][Tt][Rr][Yy]                    { return ENTRY; }
45  [Ii][Dd][Ll][Ee]                        { return IDLE; }
46  [Ee][Xx][Ii][Tt]                        { return EXIT; }
47  [Tt][Rr][Aa][Nn][Ss][Ii][Tt][Ii][Oo][Nn]  { return TRANSITION; }
48  [Aa][Cc][Tt][Ii][Oo][Nn]                { return ACTION; }
49  [Tt][Ee][Rr][Mm][Ii][Nn][Aa][Tt][Ee]    { return TERMINATE; }
50
51
52    /* identifiers */
53
54  [a-zA-Z_][a-zA-Z_0-9]*          {
55    if ( strlen(yytext) > 256 )
56      printf(
57      "%d: truncating symbol '%s' to 256 characters\n",
58      lineno, yytext);
59    strncpy(yylval.string, yytext, 256);
60    return IDENTIFIER;
61                                  }
62
63
64    /* constants */
65
66  [0-9]+\.?[0-9]*                 {
67    yylval.constant = (float)atof(yytext);
68    return CONSTANT;
69                                  }
70
71
72    /* newlines, etc. */
73
74  \n                              { lineno++; }
75  .                               ; /* ignore */
76
77
78  %%
79
80  int yywrap()
81  {
```

```
82          return 1;
83  }
84
85  void yyerror(char* s)
86  {
87          printf("\n%d: %s at %s\n", lineno, s, yytext);
88  }
```

Listing C.1: hsmc.l (lex source)

C.2 yacc, parser

```
1   %{
2   extern int yylex();
3   extern void yyerror(char* s);
4
5   #include "main.h"
6   %}
7
8   %union
9   {
10          char  string[258];
11          float constant;
12  }
13
14  %token MACHINE
15  %token STATE
16  %token DEFAULT
17  %token ENTRY
18  %token IDLE
19  %token EXIT
20  %token TRANSITION
21  %token ACTION
22  %token TERMINATE
23
24  %token <string> IDENTIFIER
25  %token <constant> CONSTANT
26
27  %%
28
29  machines: machine
30          | machines machine
31          ;
32
33  machine: machine_decl '{' state_items '}'
34           {
35             parseEndMachine();
36           }
37         ;
38
```

```
39 state_items: state_item
40             | state_items state_item
41             ;
42
43 state_item: state
44           | statement
45           ;
46
47 state: state_decl '{' state_items '}'
48        {
49          parseEndState();
50        }
51      ;
52
53 statement: default ';'
54          | entry ';'
55          | idle ';'
56          | exit ';'
57          | transition ';'
58          | action ';'
59          | timetransition ';'
60          | timeaction ';'
61          | terminate ';'
62          ;
63
64 machine_decl: MACHINE '(' IDENTIFIER ',' CONSTANT ')'
65               {
66                 parseBeginMachine($3,(int)$5);
67               }
68             ;
69
70 state_decl: STATE '(' IDENTIFIER ')'
71             {
72               parseBeginState($3);
73             }
74           ;
75
76 default: DEFAULT '(' IDENTIFIER ')'
77          {
78            parseDefault($3);
79          }
80        ;
81
82 entry: ENTRY '(' IDENTIFIER ')'
83        {
84          parseEntry($3);
85        }
86      ;
87
88 idle: IDLE '(' IDENTIFIER ')'
```

```
89          {
90            parseIdle($3);
91          }
92        ;
93
94  exit: EXIT '(' IDENTIFIER ')'
95          {
96            parseExit($3);
97          }
98        ;
99
100 transition: TRANSITION '(' IDENTIFIER ',' IDENTIFIER ')'
101             {
102               parseTransition($3,$5);
103             }
104           ;
105
106 action: ACTION '(' IDENTIFIER ',' IDENTIFIER ')'
107         {
108           parseAction($3,$5);
109         }
110       ;
111
112 timetransition: TRANSITION '(' CONSTANT ',' IDENTIFIER ')'
113                 {
114                   parseTimeTransition($3,$5);
115                 }
116               ;
117
118 timeaction: ACTION '(' CONSTANT ',' IDENTIFIER ')'
119             {
120               parseTimeAction($3,$5);
121             }
122           ;
123
124 terminate: TERMINATE '(' IDENTIFIER ')'
125            {
126              parseTerminate($3);
127            }
128          ;
```

Listing C.2: hsmc.y (yacc source)

Glossary

blah A blah is a blah blah blah!

Bibliography

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Sam02] Miro Samek. *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems.* CMP Books, 2002.

Index