Intuitive
Hierarchical State Machine
Programming

HSM Notation Language Specification version 1.0

16 July 2016, hsmp version 1.0

Victor Li

vli02@hotmail.com

Introduction

hsmp defines an Intuitive Notation Language for writing a UML Hierarchical State Machine program. By using this language, a hierarchical state machine is composed as a Grammar File. hsmp takes this grammar file and generates a high efficient hierarchical state machine in C code.

The overall grammar of hsmp is very much yacc or Gnu Bison alike but simpler and intuitive, brings no hassle for an experienced computer programmer to learn and use.

hsmp uses UML state machine as the standard definition of Hierarchical State Machine, it supports UML state machine version 2.0.

The target code hsmp v1.0 generates is in C language. Future releases may support to generate code in Java, C++ and other languages. Generating a hierarchical state machine diagram may be supported in future releases, too.

Conditions for Using hsmp

hsmp and its associated documentations are the intellectual property of the author. They are free for evaluation purpose or noncommercial use, a license is required for commercial production use.

Contact the author for commercial license information at:
vli02@hotmail.com or
vli02us@gmail.com

Table of Content

1. Examples

1.1. A simple hierarchical state machine grammar:

```
%{ /* prologue */ }%

%event a b c x
%state s0 s1 s2 s21
%hiera s0 (s1, s2 (s21))

%%
s0: x --
;

s1: a -> s2
;

s2: b -> s1
;

s21: c -> s0
;

%%

/* epilogue */
```

This example declares a list of event `a`, `b`, `c` and `x`; and a list of state `s0`, `s1`, `s2` and `s21`.

The state `s0` has sub states `s1` and `s2`; the state `s2` has sub state `s21`.

On state `s0`, it performs an internal transition when event `x` is received.

On state `s1`, it transition to state `s2` if event `a` is received.

And so on…

1.2. A simple example grammar with transition actions:

```
%{
#include <stdio.h>
int test = 1;
}%
%event a b c x
%state s0  -> { printf("s0-entry"); }
           <- { printf("s0-exit"); }
       s1  -> { printf("s1-entry"); }
           <- { printf("s1-exit"); }
       s2  -> { printf("s2-entry"); }
           <- { printf("s2-exit"); }
       s21 -> { printf("s21-entry"); }
           <- { printf("s21-exit"); }
```

```
%hiera s0 (s1, s2 (s21))

%%

s0: x --    { exit(0); }
;

s1: a -> s2 { printf("s1-a-s2"); }
;

s2: b -> s1 { printf("s2-b-s1"); }
;

s21: c ?(test == 1) -> s0
            { printf("s21-c-s0"); }
;

%%

/* epilogue */
```

Every state has an entry and exit function which prints an information.

State `s0` performs internal transition on event `x`, the transition action is to exit the program.

State `s1` and `s2`'s transition have transition actions which print the source state, event and destination state names.

Transition of state `s21`, on event `c`, to destination state `s0` has a guard condition, which is to test expression "`(test == 1)`", the transition is enabled if the guard condition is evaluated true, the transition action is to print source state, event and destination state names.

2. Invoking hsmp

The simple way to invoke hsmp is:

hsmp *input_file*

Here *`input_file`* is the name of a state machine grammar file, which usually ends in `'.def'`. The name of generated C file is made by appending `'.tab.c'` to the input file name and in the same directory of the input file.

By default, hsmp prints version and copy right information.

2.1. hsmp options:

`-h, --help`

Print a summary of help information.

`-o <file>, --output <file>`

Specify an output file name.

`-d, --define`

Produce a header file.

`-v, --verbose`

Produce a '.output' file.

`-n, --no_lines`

Do not generate '#line' directives.

`-S, --Strings`

Generate string tables and related utility functions for all events and states.

`-W <n>, --Warning <n>`

Specify warning level, default warning is on highest level.

`-i, --silent`

Hide version and copy right information.

`-V, --version`

Print version information.

`-L, --license`

Print license information, this overrides `-i` option.

`-A, --add_license`

Apply a license to the software.

3. Grammar File

As it is shown in the examples, a grammar file contains four main sections, shown here with appropriate delimiters:

```
%{
   prologue
%}
declaration section
%%
transition rules section
%%
epilogue
```

hsmp grammar supports C style line and block comments like:

```
// line comments

/*
 * block comments
 */
```

3.1. The Prologue

The prologue section contains a piece of code in C, the code can be header file inclusion, function prototypes, variables or macros definitions, or anything in C code.

The C code in placed inside clause `'%{ … }%'`.

Example of a prologue:

```
%{
#include <stdio.h>       // header file
unsigned int foo();      // function prototype
int test = 0;            // global variable
}%
```

In the generated file, the prologue code will appear in the very beginning place of the file.

3.2. Declaration Section

Followed by the prologue, the declaration section contains the declarations of major elements of a hierarchical state machine, i.e.: events, states and state hierarchies.

Beside the major elements, there are several miscellaneous declarations.

Each declaration starts with notation `'%'` and a keyword.

3.2.1. Event declaration

```
%event <a list of events separated by space>
```

The list of events can be one or more events, they can be in one line or in multiple lines.

More than one event declaration can be used in a grammar file.

Example of event declarations:

```
%event a
%event b c d
%event x
       X
```

In the generated file, each event will be defined as a C macro and assigned an numeric ID to it if manual event ID is not declared (see section 3.2.4). If option `'-S'` is applied to hsmp, it will also produce an array containing the literal string of each event.

An event declaration is optional. If it is omitted, hsmp will automatically declare the event if it is used in subsequent sections. Such is called as implicit declaration. A warning message will be printed for this implicit declaration when hsmp generates the state machine code.

3.2.2. State declaration

```
%state <a list of states separated by space>
```

Similar to event declaration, this is to declare state element of a state machine. The list of state can contain one or more states in one or multiple lines.

More than one state declarations can be used in a grammar file.

Example of state declaration:

```
%state s0
%state s1 s2
          s21
```

In the generated file, each state will be defined as a C macro and assigned an numeric ID to it if manual state ID is not declared (see section 3.2.4). If option `'-S'` is applied to hsmp, it will also produce an array containing the literal string of each state.

State declaration is optional, too. If it is omitted, hsmp will automatically declare the state if it is used in subsequent sections. Such is called as implicit declaration. A warning message will be printed for this implicit declaration when hsmp generates the state machine code.

A state can have an optional entry action and an optional exit action. The entry and exit action declarations are followed in a state in the state declaration. An implicitly declared state has no entry and exit actions.

Entry action uses notation:

```
‘-> {‘ action ‘}’
```

where `action` is user code, it is usually C statement.

Similarly, exit action uses notation:

```
‘<- {‘ action ‘}’
```

Example:

```
%state s0  -> { printf(“s0-entry”); }
           <- { printf(“s0-exit”); }
%state s1  -> { printf(“s1-entry”); }
       s2  <- { printf(“s2-exit”); }
       s21
```

where s0 has both entry and exit actions, s1 has entry action only and s2 has exit action only, s21 does not have entry and exit actions.

3.2.3. State hierarchy declaration

```
%hiera <state hierarchies>
```

State hierarchies is expression of super-sub states with format:

```
<a state> ‘(‘ <a list of states separated by comma> ’)’
```

Where the leading state before the pair of parenthesis is a super state, and the list of states inside the pair of parenthesis is sub states of the super state.

More than one level of super-sub state hierarchies can be nested in one hierarchy declaration or separated into multiple hierarchy declarations.

One super state can have only one state hierarchy declaration.

A state which does not appear as a super state in any state hierarchy declarations is called a leaf state.

A leaf state can have no more than one super state.

Example of state hierarchy declarations:

```
%hiera s0 (s1, s2 (s21))
```

or

```
%hiera s0 (s1, s2)
%hiera s2 (s21)
```

3.2.4. Miscellaneous declarations

a) Start state declaration

```
%start <a state>
```

This declares the start state of a state machine. It is optional. If it is omitted, the start state will be the first source state appears in the transition rules section. Such is called implicit start state declaration. hsmp reports a warning for implicit start state declaration by default.

The start state can associate with a start transition action. When state machine starts to run, it will execute the start transition as the first step and then set current active state to the start state.

A start transition is optional. Implicit start state declaration has no a start transition action.

Example:

```
%start s0 { /* start transition action */ }
```

b) Prefix declaration

```
%prefix <identifier>
```

This declares a prefix for the names of interfaces in hsmp generated state machine code. The interfaces include a data structure, all non static functions and certain macros.

The prefix will be converted to all capital for macros.

This is to provide convenience in order to avoid duplicated symbols if a program contains more than one hierarchical state machine design with more than one grammar files.

Example:

```
%prefix k2_
```

c) Manual event and state ID declarations

`%manual_event_id`

Using this declaration to tell hsmp not to automatically define event ID macros, instead, use the event ID manually defined by user.

`%manual_state_id`

Using this declaration to tell hsmp not to automatically define state ID macros, instead, use the state ID manually defined by user.

3.3. Transition Rules Section

This section is right after the first delimiter `'%%'`. It defines a set of rules for some states to react based on certain events being received. The reaction is the transitions which are one or more possible internal transitions, external transitions or local transitions to a different state.

The guard condition is following after event in a possible transition, the transition action is following after the transition type or destination state if there is one.

The transition attempting is from top to bottom and left to right, attempting stops once a transition is qualified and performed.

Grammar of transition rules:

```
<a source state> ':' [events] transitions
                    [[events] transitions] …
';'
```

Transition rules declaration starts with a source state, followed by a column, events-transitions pairs come after, and it ends with a semi-column. It is very much yacc or Gnu Bison grammar alike.

One state can have one or more than one events-transitions pairs, more than one events-transitions pairs can be declared in one transition rules declaration by following one pair to another, they can also be declared in multiple transition rules declarations separately, i.e. one source state can have multiple transition rules declarations.

The list of events can be one or more events separated by comma `','`, an optional pair of parenthesis can be used for the list of events. When multiple events are used in one events-transitions pair, it means the same transition apply to any one of these events being received.

Grammar of `transitions` is:

```
 [guard] <transition type> [state] [action]
[[guard] <transition type> [state] [action]] …
```

As it is shown here, one or more than one possible transitions can be followed one by one. For those rules having more than one possible transitions, the guard is used to enable no more than one destination for the transition of the source state on the event being received in a transition rules declaration.

3.3.1. Guard condition

A guard condition is expressed by notation:

```
‘? (‘ expression ‘)’
```

Inside the pair of parenthesis is a logical expression, it will be evaluated as true or false. If a guard condition is omitted, the transition is always evaluated as true.

The transition is enabled if guard condition is evaluated as true, and disabled if it is evaluated as false.

Example:

```
? (test == 1)
```

3.3.2. Transition type

A transition type is one of followings:

Type	Notation
Start transition	`N/A`
Initial Transition	`. ->`
Internal Transition	`--`
External Transition	`->`
Local Transition from a sub state to it's super state	`>|`
Local Transition from a super state to it's sub state	`|>`

Start transition has no a notation, see section 3.2.4.

3.3.3. Initial transition

An initial transition of a super state to its sub state has no an event. Notation dot `'.'` is used here to simulation of initial transition in the form of UML state machine diagram.

If an initial transition of a super state is not declared, hsmp will implicitly declare an initial transition to the first sub state appearing in the super state hierarchy declaration, and without an initial transition action. hsmp reports a warning for such implicit initial transition declaration by default.

Initial transition has no a guard condition.

3.3.4. Internal transition

Internal transition has no a destination state, all the rest of transition types including initial transition must have a destination state.

Internal transition only executes the transition action, it does not involve any exit or entry actions and introduce state change.

3.3.5. External transition

External transition always involve exit actions, transition action and entry actions, and transition to destination state.

If destination state is the same state of source state, such a transition is also called self transition.

All actions will be executed in the order of exit actions in the first, then transition action and entry actions in the last.

Exit actions will be executed in the order of from the innermost state to the outermost state.

Entry actions will be executed in the order of from outermost state to the innermost state.

3.3.6. Local transition

Lock transition is similar with external transition, but not run exit and re-entry of the super state.

3.3.7. Transition action

Similar to a state's entry or exit action, a transition action is expressed by notation:

```
'{' action '}'
```

where `action` is user code which is usually C statements. If a transition action is omitted, there will be additional action to be executed when the state transition happens.

Example 1:

```
s0: . -> s1 ;
s0: x -- ;

or

s0: . -> s1
    x --
;
```

Where `s0` has an initial transition to it's sub state `s1` and internal transition on event `x`; both do not have an transition action.

Example 2:

```
s2: . -> s21
    b, B -> s1
;

or

s2: . -> s21
    (b, B) -> s1
;
```

Where `s2` has an initial transition to it's sub state `s21` and one external transition to state `s1` when event `b` or `B` is received.

Example 3:

```
s0: . -> s1          { printf("s0-s1 init"); }
    x --             { exit(0); }
;
s1: a -> s2          { printf("s1-a-s2"); }
;
s2: . -> s21         { printf("s2-s21 init"); }
    (b, B) ?(test == 1) -> s1
                     { printf("s2-%s-s1", evt_str); }
                          -> s2
                     { printf("s2-%s-s2", evt_str); }
    d |> s21         { printf("s2|d—s21"); }
;
s21: c -> s0         { printf("s21-c-s0"); }
     e >| s2         { printf("s21-e|s2"); }
;
```

In additional to previous two examples, this example has transition actions for every transition rule which is to print the transition information or exit the program.

Source state `s2` on event `b` or `B` has two possible transitions, when event `b` or `B` is received, it firstly attempts on the first one which has a guard condition `(test == 1)`.

If the condition is evaluated true, the transition to destination state `s1` will be enabled and performed; Otherwise it will continue to attempt on the second one which has no a guard condition so it will be always evaluated true and transition to itself will be performed.

The `evt_str` in the transition action is a global variable of string type provided by user code.

State `s2` has a local transition to it's sub state `s21` on event `d`.

State `s21` has a local transition to it's super state `s2` on event `e`.

3.4. The Epilogue

This section is after the second delimiter `'%%'`. Everything in this section will be user code.

In the generated C file, all content of this section is placed in the last of the file.

Example:

```
%%

void foo() { /* c code */ }
int main() { /* run the state machine */ }
```

4. C-Language Interfaces

In the hsmp generated C and/or header files, there are one data structure, several functions and macros. All these interfaces together represent a hierarchical state machine defined by the grammar file. They will be used in a user program in order to create the state machine, run it and get state information from it.

Most of interfaces are with a prefix which was declared in the declaration section by using `%prefix` keyword.

4.1. Data Structure

```
typedef struct <prefix>hsm_s {
   unsigned int (*hh_event_func)(void *,
                                 void **);
   void         (*hh_deferral_func)(void *,
                                    unsigned int,
                                    void *,
                                    int);
   void         (*hh_lock_func)(void *);
   void         (*hh_unlock_func)(void *);
   void          *hh_private_data;
   int            hh_state_id[n];
   int            hh_running;
} <prefix>hsm_t;
```

An instance of this data structure type is the instance of a state machine defined by the grammar file.

The instance can be statically allocated or dynamically allocated by using malloc. Multiple instances of a state machine can be created by creating multiple instances of this data structure type.

Examples:

```
k2_hsm_t  my_k2;
k2_hsm_t *my_k2_p = (k2_hsm_t *)
                     malloc(sizeof(k2_hsm_t));
…
free(my_k2_p);
```

where `my_k2` is a statically allocated `k2` state machine instance;
`my_k2_p` is a pointer being dynamically allocated and points to a `k2` state machine instance, it is freed in the end of the program.

Before an instance of a state machine can start to run, there is a need to initialize the instance.

4.2. Initialization Function

```
int <prefix>hsm_init(
   <prefix>hsm_t *,
   unsigned int (*)(void *, void **),
   void (*)(void *, unsigned int, void *, int),
   void (*)(void *),
   void (*)(void *),
   void  *);
```

This function is used to initialize a state machine instance before running the state machine.

If successful, the function returns value 0; otherwise -1.

4.2.1. The first argument is a pointer to a state machine instance which is being initialized. The pointer has to point to a valid non empty state machine instance.

4.2.2. The second argument is a pointer to a function which is used to receive events in a state machine runtime. It's user's responsibility to manage event queuing or dispatching system and provide this function as a callback for hsmp generated state machine to receive event one at a time.

This argument has to be a pointer points to valid event receiving function, a NULL will cause the initialization to fail.

An event is an `unsigned int` type ID should be returned from the event receiving function. When this function is called, it passes a pointer which points to the state machine associated private data, and a pointer of a pointer which is for the output argument of the function if there is an event data associated to the event ID being returned.

The event ID and event data can be accessed thru hsmp generated macro interfaces in the entry, exit or transition actions in state machine runtime.

4.2.3. The third argument is a pointer to a function which is used to defer an event when event deferral is used in a grammar file and an event is being deferred. See advanced.

If event deferral is not used in the grammar file, a NULL can be used for the initialization.

4.2.4. The fourth and fifth arguments are a pair of pointers of lock and unlock function. If the state machine instance being initialized in the first argument might be accessed simultaneously in more than one threads, there is a need to provide a lock and unlock functions which will be used to protected the state machine instance when a transition is happening in the state machine runtime.

It is user's responsibility to make sure appropriate lock and unlock is provided if the instance might be accessed simultaneously in different threads. If the instance is accessed in one thread only, NULLs can be used for the initialization.

4.2.5. The last argument is a pointer points to a private data the state machine instance is associated with. This private data is passed to the calling of event receiving and deferral functions being initialized in the second and third arguments respectively. It can be accessed thru hsmp generated macro interfaces in the start, initial, entry, exit or transition actions in state machine runtime, too.

If there is no a private data associated with a state machine instance, a NULL can be used for the initialization.

Examples:

```
unsigned int read_key(void *, void **);
int main()
{
    k2_hsm_t  my_k2;
    rc = k2_hsm_init(&my_k2, read_key,
                     NULL, NULL, NULL, NULL);
    if (rc != 0) {
        return -1;
    }
    …
}
```

Which initializes the state machine instance `my_k2` with a valid event receiving function only and the rest as NULLs.

4.3. Running Function

```
int <prefix>hsm_run(<prefix>hsm_t *);
```

This is the function to run a state machine instance, as being passed as the argument, from scratch. It will firstly perform the start transition to

the start state being explicitly or implicitly declared in the grammar file, then fall into a main loop of receiving an event and processing the event.

The loop stops and returns with value 0 or -1 when it is directed to do so by using hsmp generated macros `HHACCEPT` or `HHABORT` in an entry, exit or transition action. See section 4.9.

It receives event by calling the event receiving function which is provided in the initialization of the state machine instance.

The process of an event is either of the following ways:

4.3.1. Event is invalid and will be ignored after notifying user thru a hsmp generated macro.

4.3.2. Event is valid but not being handled, or being handled but none of guard conditions is satisfied on current leaf and all super states in the state machine. The event is ignored after notifying user thru a hsmp generated macro.

4.3.3. Event is deferred by current leaf or one of super states in the state machine. The event deferral function will be called with the event ID and associated event data being passed to.

4.3.4. A transition will be performed. It is either an internal transition, external transition or local transition. The related exit actions of

source states, transition action and entry actions of destination states will be executed in the order being defined by UML state machine. The event ID and associated data can be accessed in these actions thru hsmp generated macros.

If lock and unlock functions are provided in the state machine instance initialization, the lock function will be invoked before transition starts and unlock function will be invoked after transition is competed.

4.4. Get State Function

int <prefix>hsm_get_state(<prefix>hsm_t *);

This function can be used to get the ID of current leaf state of a state machine instance.

This function calls the lock and unlock functions if they are provided in the state machine instance initialization. So the function may be blocked if state machine is in the middle of running a transition.

4.5. String Utility Functions

```
const char *<prefix>hsm_event_str(unsigned int);
const char *<prefix>hsm_state_str(int);
```

If hsmp is invoked with option -S or --String, these two function will be generated. They are for converting an event or state ID to event or state literal string respectively. They are particularly useful for logging and debugging purpose.

4.6. Event and state ID macros

If `%manual_evnet_id` is not declared in the grammar file, hsmp will automatically defines all events as macros and assigned an ID to each event.

If `%manual_state_id` is not declared in the grammar file, hsmp will automatically defines all states as macros and assign an ID to each state.

Otherwise, user has to define event or state IDs and provide them for hsmp generated code to use.

4.7. Macros <PREFIX>NUM_OF_EVENTS and <PREFIX>NUM_OF_STATE

hsmp generates these two macros with the value of number of event and state being declared in the grammar file.

4.8. Macros HHTRUE and HHFALSE

These two macros are true and false of a hsmp generated state machine evaluates guard condition as, so all guard conditions should these two macros for the true and false conditions.

4.9. Macros HHACCEPT and HHABORT

These two macros can be used in an entry, exit or transition action function. When it is invoked during the state machine is running, it will cause to terminate the main loop of the state machine run function, i.e.:

HHACCEPT will cause function <prefix>hsm_run(…) to return with value 0.

HHABORT will cause function <prefix>hsm_run(…) to return with value -1.

Example:

```
s0: . -> s1        { printf("s0-s1 init"); }
    x --           { HHACCEPT; }
    X --           { HHABORT; }
;
```

4.10. Macro HHPDATA

This macro is used to access a state machine instance private data in a state machine runtime. The private data is the one being initialized to the state machine instance thru the function <prefix>hsm_init(…).

It can be used in any places of a state machine runtime activities. State machine runtime activities are defined as:

Entry, exit and transition actions, guard condition, and definition of the following macros:

```
HH_PROCESS_…_EVENT
HH_TRACE_…
```

Example:

```
s1: . -> s11 { foo(HHPDATA); }
;
```

4.11. Macros HHEVENT and HHEVTARG

They are the event ID and pointer of event data associated to the event ID being received from event receiving function in a state machine runtime. They can be used in any places of a state machine runtime activities (see section 4.6).

Example:

```
s1: . -> s11 { foo(HHPDATA); }
    a -> s1  { bar(HHEVENT, HHEVENTARG); }
;
```

4.12. Macros HHSSTATE, HHESTATE, HHDSTATE and HHCSTATE

These macros reflect to the source, effective, destination and current state ID in a state machine instance where a transition is happening. They can be used in entry, exit and transition actions, and macro HH_TRACE_…

The source state is always the source leaf state ID where the transition is taking place.

The effective state is the leaf or one of it's super state ID which the transition applies to.

The destination state is the state ID where the transition is going to. In a internal transition which has no a destination state, behavior of using macro `HHDSTATE` is undefined.

All these three state IDs won't change in a whole transition process, but the current state ID changes when transition is in processing.

The current state ID always reflects to the current state when an entry, exit or transition action is performed, it changes thru the most inner source state, thru all super states, to the destination state and down to the most inner final leaf state.

Example:

```
s1: . -> s11 { foo(HHPDATA); }
    a -> s1  { bar(HHEVENT, HHEVENTARG); }
    b -> s11 { goo(HHSSTATE, HHDSTATE); }
;
```

4.13. Macros `HH_PROCESS_INVALID_EVENT` and `HH_PROCESS_IGNORED_EVENT`

These two macros are hooked in the place of a generated state machine where an event is received, and being determined invalid or ignored. They are defined as empty by default.

They can be redefined to a piece of user code, if so, this piece of code will be invoked when such an event is received in state machine runtime.

Example:

```
#define HH_PROCESS_INVALID_EVENT \
printf("invalid event: %d.\n", HHEVENT)

#define HH_PROCESS_IGNORED_EVENT \
printf("event %s is ignored on state %s.\n", \
        <prefix>hsm_event_str(HHEVENT),\
        <prefix>hsm_state_str(HHSSTATE))
```

4.14. Macros `HH_TRACE_STRAT`,
`HH_TRACE_EXIT`,
`HH_TRACE_ACTION`,
`HH_TRACE_ENTRY`,
`HH_TRACE_INIT` and
`HH_TRACE_TRANS(END)`

These macros are hooked in the place of a generated state machine after certain transition action is done. They are defined as empty by default. They can be redefined to a piece of user code, if so, this piece of code will be invoked when certain activity is done in state machine runtime.

Example:

```
#define HH_TRACE_START \
   printf(“start-INIT”)
#define HH_TRACE_ENTRY \
   printf(“%s-ENTRY”, \
          <prefix>hsm_state_str(HHCSTATE))
#define HH_TRACE_EXIT \
   printf(“%s-EXIT”, \
          <prefix>hsm_state_str(HHCSTATE))
#define HH_TRACE_INIT \
   printf(“%s-INIT”, \
          <prefix>hsm_state_str(HHCSTATE))
#define HH_TRACE_ACTION \
   printf(“%s-%s-%s”, \
          <prefix>hsm_state_str(HHESTATE), \
          <prefix>hsm_event_str(HHEVENT), \
          (HHDSTATE != -1) ? \
          <prefix>hsm_state_str(HHDSTATE) : “”)
#define HH_TRACE_TRANS(LAST) \
   printf(“%s-%s-%s %s”, \
          <prefix>hsm_state_str(HHSSTATE), \
          <prefix>hsm_event_str(HHEVENT),  \
          <prefix>hsm_state_str(HHCSTATE), \
          (LAST) ? "ALL DONE" : "DONE")
```

5. Advanced Features

In addition to all features above, certain features being provided by hsmp are considered as advanced as they involves in some more complicated hierarchical state machine design.

5.1. Event Deferral

hsmp does not generate and provide event queuing or dispatching code, it's user's responsibility to manage it and provide a callback function to hsmp generated state machine code thru a state machine instance initialization for receiving event at state machine runtime.

Deferring an event is basically to send the event back to event queuing or dispatching system for holding it and make it being received again once a state change happens in the state machine.

There are two ways to support event deferral in hsmp:

5.1.1. Event Deferral Transition Rule

hsmp uses a notation `'<<'` in state transition rule to specify the events being deferred. This rule has no a guard and action associated with it.

If this rule is used in a grammar file, the initialization of the state machine instance requires the event deferral callback function. The initialization will fail if this callback function is not provided.

When this rule transition is ran in state machine runtime, the event deferral callback function will be invoked with parameters of the event ID and data which is deferred. It's user's responsibility to handle this event so it can be received once a state change happens in the state machine.

Example:

```
s2: . -> s21
    a <<          /* event deferral */
    c -> s1
    f -> s11
;
```

5.1.2. Deferral Thru Internal Transition

There is nothing special here with a normal internal transition rule. The event is actually taken and consumed by the transition action. The action can send the event back to user's event queuing or dispatching system.

This is a natural way to support event deferral. As a normal internal transition, a guard condition can be supplied and any complicated transition actions can be taken beside sending the event back, so it has some more flexibilities.

Example:

```
s2: . -> s21
    a --   { /* enqueue ‘a’ to deferred queue */ }
    c -> s1
    f -> s11
;
```

5.2. Wild-character Event

hsmp internally defines a wild character event with notation '*'. It can be used in a transition rule, means any valid event being received applies to the transition rule.

Wild character event has no difference than a normally declared event. A guard condition can be used for the transition rule, and the transition is only enabled if the guard condition is evaluated true.

Wild character event has no an ID. The real ID of the event which triggered the transition can be known thru macro `HHEVENT` in the exit, transition and entry actions of the transition.

Example:

```
s0: . -> s1
    e -> s211
    x ->      { HHACCEPT; }
    X ->      { HHABORT; }
    * --      { printf("%s matched *\n",
                <prefix>hsm_event_str(HHEVENT)); }
;
```

5.3. Orthogonal Regions

hsmp supports orthogonal regions and introduces two new terms Zone and Region. These two terms only apply to all documentations, and hsmp warning and error messages. They are normal states and don't introduce any new thing in grammar file.

A super state which has more than one regions is called a Zone state. Each direct sub state of a Zone state, i.e. the first level of sub states, is called a Region state.

A Zone state is recognized by declaring an initial transition which transitions to each of it's Region state.

All Region states inside a Zone have to be independent from each other. No state inside a Region can transition to the outside of the Region itself.

The exit of a Region state can only be made on it's parent Zone state which will cause exit from all Region states.

Local transition between a Zone state and any of it's Region state is prohibited. This is to avoid yielding duplicated active regions.

Example:

```
%state z1 r1 r2 s1 …
%hiera z1 (r1, r2)
…
%%
…
z1: . -> r1
    . -> r2
    c -> s1
;
```

Where state `z1` is a Zone state and it has two Region states `r1` and `r2`. When a transition to state `z1`, two regions `r1` and `r2` will be formed and simultaneously active. A subsequent event `c` will cause these two regions and zone states gone and transition to state `s1`.

Followings are additional interfaces being generated for Orthogonal Regions feature.

5.3.1. Get All States

```
unsigned int
<prefix>hsm_get_all_states(<prefix>hsm_t *,
                           int *,
                           unsigned int);
```

In addition to get state function, get all state function take two more arguments, one integer type of array for the output of state IDs of all current active Regions, and the length of array being passed in.

The return value is the number of state IDs being returned thru the array.

If more than one regions are currently active, get state function will return any one of the state ID of current active regions. If there is only one of active region, get state and get all state functions have the same effect.

5.3.2. Macro <PREFIX>MAX_REGIONS

This macro is the maximum number of active regions a state machine defined by grammar file can be formed at a time in state machine runtime.

The value of get all states function returns won't be greater than this number.

5.3.3. Macro `HHRINDEX`

This is an internal index of hsmp generated state machine. It can be accessed in exit, transition and entry actions of a transition, and all `HH_TRACE_...` macros.

5.3.4. Macro `HH_REGION_GONE(I)`

If a transition will result in a region to be gone, this macro is invoked in the last exit action of the transition with parameter of current region index. It is defined as empty by default and can be redefined to a piece of user code.

A new region might be formed and can be caught in macro `HH_TRACE_TRANS(LAST)`. The parameter `LAST` indicates whether this transition is the last one of all active regions.

5.4. Run To History

```
int
<prefix>hsm_run_to(<prefix>hsm_t *,
                   const int *,
                   unsigned int);
```

hsmp generated state machine supports to start running directly from history states, instead of from scratch which will run start transition to the start state.

In addition to the run function, it takes an integer type array as the input of last active history state IDs. If there is only one active region in the history, this array contains only one history state. The last argument is the number of input state IDs in the input array.

Example:

```
k2_hsm_t my_k2;
int rc, state;
…
rc = k2_hsm_run(&my_k2);
…
state = k2_get_state(&my_k2);
…
rc = k2_hsm_run_to(&my_k2, &state, 1);
…
```

6. Appendix A Notations

`%{ code }%`

Prologue grammar. All code between '%{' and '}%' are copied verbatim to the generated state machine implementation file.

`%`

Leading symbol of a keyword. e.g. `%event`, `%state`, `%hiera`, etc…

`%%`

Section delimiter in between of declaration, transition rules and epilogue sections.

`->`

If it is used in a state declaration, it is entry function notation.
if it is used in a state transition rule, it is external transition notation.

`<-`

Exit function notation, it is used in a state declaration.

`.->`

Initiation notation, it is used in a super state transition rule.

`--`

Internal transition notation.

`|>`

Notation of a local transition from a super state to one of it's sub states.

`>|`

Notation of a local transition from a sub state to it's super state.

`<<`

Event deferral notation.

? (expression)

Notation of a guard condition in a transition rule. **expression** is logically evaluated as HHTRUE or HHFALSE.

{ code }

Action notation. It is either entry or exit action of a state, or a transition action. **code** will be executed when the action is invoked in a state transition.

:

Delimiter between source state and event(s) in transition rule gramma.

*****

Wild-character event notation, it is for use in state transition rule.

;

End of a declaration of a state transition rule.

7. Appendix B Keywords

%prefix

Declare a prefix for generated data structure, function name and some of Macro names in a state machine implementation file(s).

%manual_event_id

Declare using manual defined event IDs instead of automatically generating event ID definitions in a state machine implementation file(s).

%manual_state_id

Declare using manual defined state IDs instead of automatically generating state ID definitions in a state machine implementation file(s).

%event

Declare one or a list of event.

%state

Declare one or a list of event with optional entry and exit actions.

%hiera

Declare super-sub states hierarchies.

%start

Declare start state of a state machine.