

NAME

CHSM-C++ – Concurrent, Hierarchical, Finite State Machine specification language for C++

SYNOPSIS

```

declarations
%%
description
%%
user-code

```

DESCRIPTION

The CHSM specification language is a text-based means for specifying *statecharts*. A statechart is graphical formalism for specifying concurrent, hierarchical, finite state machines. Such machines are useful for developing reactive systems. Additionally, arbitrary C++ code can be integrated in a manner similar to the way `yacc(1)` allows integration of arbitrary C code.

LEXICAL CONVENTIONS**File Format**

A CHSM description file is an ordinary text file with three sections: declarations, description, and user-code. The sections are separated by the `%%` token that must be at the beginning of a line. The declarations and user-code sections may be empty; if the user-code section is empty, then the trailing `%%` may be omitted. If present, the declarations and user-code sections are passed through, untouched, to the underlying C++ compiler. The description file is otherwise free-format. The smallest legal CHSM description file is:

```

%%
chsm smallest is { }

```

The declarations section is meant to contain any C++ declarations or definitions, or preprocessor directives needed to compile the resultant C++ code.

Comments

Both `/* ... */` and `//` comments may be used anywhere in the CHSM description file where white-space is legal.

Identifiers

Identifiers follow the same rules for what constitutes a valid identifier in C++.

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

```

chsm      cluster  deep  enter  event  exit
history  in        is    set   state  upon

```

Additionally, all C++ keywords are also reserved.

The following characters and character-combinations are used as punctuation or operators:

```

,   ;   ->  .   ::  %%  %{  %}  $
[   ]   (   )   {   }   <   >   *   &

```

BASIC CONCEPTS**Scope**

There are four kinds of scope: global, CHSM, local, and code-block.

Global scope exists in the declarations and user-code sections. The only thing injected into the global scope from the description section is the name of the CHSM.

CHSM scope exists inside the body of the CHSM description. State and event names are local to CHSM scope.

Local scope exists only within clusters and sets and only for child-state identifiers. One identifier may be hidden by another due to a child-state declaration with an equal name in a lexically-enclosed

scope. Such an identifier can still be referenced by using one of the scope-resolution operators (see STATE NAMES).

Code-block scope exists only within `%{ %}` pairs. It is treated exactly like a C++ function. All state names and all event names are in the scope of all blocks. Code blocks follow the “C++ rewriting rule” for `inline` functions (see ARM, 9.3.2), although they are not `inline` functions.

Initialization

A CHSM is fully initialized upon definition. States are not automatically entered, however; nor are they automatically exited upon destruction.

```
%%
chsm my_machine is {
    // ...
}

%%
int main() {
    my_machine m;    // fully initialized and ready for use
    m.enter();      // enter initial states
    // ...
    m.exit();       // exit all states
}
```

Types

There are five fundamental types: machine, state, cluster, set, and event; each is an instance of the `CHSM::machine(3)`, `CHSM::state(3)`, `CHSM::cluster(3)`, `CHSM::set(3)`, or `CHSM::event(3)` class, respectively.

MACHINE DESCRIPTIONS

Machine descriptions have the form:

```
machine-desc:    chsm state-decl [(param-list)] [history-decl] [machine-block] parent-body
state-decl:      [<class-name>] identifier
param-list:     C++-function-formal-argument-declaration-list
history-decl:   [deep] history
machine-block:  { [enter-exit-blocks] }
parent-body:   is { [description-list] }
description-list: description [description-list]
description:    state-desc
                  cluster-desc
                  set-desc
                  event-desc
```

If `<class-name>` is given as part of `state-decl`, said class must have been derived (directly or indirectly) from `CHSM::machine`.

States declared in `description-list` are child states of an implicit cluster named `root`. Declaring a CHSM with `history` gives the `root` cluster a history. See `CHSM::cluster(3)` for more information.

Parameter Lists

Parameter lists are based on C++ function *formal* argument declarations (the grammar for which is specified in the ANSI/ISO C++ standard); hence, they may not be abstract:

```

chsm my_machine( int ) is {           // illegal: abstract declaration
chsm my_machine( int x ) is {       // legal

```

Additionally, they may not have default values:

```

chsm my_machine( int x = 0 ) is {    // illegal: default value

```

See **CHSM::machine(3)** for more information.

Enter-Exit-Blocks

Enter-exit-blocks are arbitrary C++ code executed upon CHSM or state entrances and/or exits. They have the form:

```

enter-exit-blocks:  enter-exit-block [enter-exit-block]

enter-exit-block:   upon enter-exit-selector %{ C++-statement-list %}

enter-exit-selector:  enter
                      exit

```

For example:

```

state s {
  upon enter %{
    cout << "hello, world\n";
  %}
  upon exit %{
    cout << "goodbye, world\n";
  %}
}

```

Enter-exit-blocks are optional. At most one of each can be specified and they must precede all transition specifications. They can be specified in either order with no semantic difference.

Within *C++-statement-list*, the variables `event` and `state` are available and are references to the event that triggered the transition and enclosing state, respectively:

```

upon enter %{
  if ( event == alpha )
    // ...
%}

```

STATE DESCRIPTIONS

Plain-state descriptions have the form:

```

state-desc:  state state-decl state-def

state-def:   state-body
            ;

state-body:  { [enter-exit-blocks] [transition-list] }

```

If `<class-name>` is given as part of *state-decl*, said class must have been derived (directly or indirectly) from `CHSM::state`. See **CHSM::state(3)** for more information.

A *state-def* of just a semicolon functions as a “sink”; such a state can be “escaped” from only by means of a transition from its parent state:

```

cluster display( normal, /* ... */, error ) {
    reset -> display; // escapes "sink"
} is {
    state normal {
        disaster -> error;
    }
    state error; // "sink"
}

```

CLUSTER DESCRIPTIONS

Cluster descriptions have the form:

cluster-desc: cluster *state-decl* (*child-list*) [*history-decl*] [*state-body*] *parent-body*

child-list: *child-name* [, *child-list*]

child-name: *identifier*

The *child-list* declares the names (only) of all child states; all child states must be declared:

```

cluster c(x,y) is {
    state x;
    state y;
    state z; // error: undeclared child
}

```

The order in which child states are declared need not match the order in which they are defined. The first child-state **defined** in the *description-list* is the default child-state.

See **CHSM::cluster**(3) for information about history.

SET DESCRIPTIONS

Set descriptions have the form:

set-desc: set *state-decl* (*child-list*) [*state-body*] *parent-body*

See **CHSM::set**(3) for more information.

EVENT DESCRIPTIONS

Event descriptions have the form:

event-desc: event [*<event-name>*] *identifier* [([*param-list*])] [*precondition*] ;

Parameter declarations for events are the same as those for machine descriptions. If *<event-name>* is given, said event is a *base event*. See **CHSM::event**(3) for more information.

Parameter Lists

If an event has been declared with parameters, they can be accessed via the C++ operator-> (not to be confused with the CHSM transition operator ->):

```

event say( char const *message );

state s {
    say -> t %{
        cout << say->message << endl;
    };
}

```

Additionally, all parameters inherited from base events, if any, are also accessible via operator->:

```

event<say> quote( char const *author );

state s {
    quote -> t %{
        cout << quote->message
            << "\n-- " << quote->author << endl;
    };
}

```

Preconditions

A *precondition* for an event is arbitrary C++ code that determines whether conditions are right for an event to be allowed to take place. Preconditions have the form:

```

precondition:  [ C++-expression ]
               %{ C++-statement-list %}

```

For example:

```

event mouse( int x, int y ) [ x >= 0 && y >= 0 ];

```

would discard all mouse events when either coordinate is negative. If C++ code for a precondition is more complicated than a simple expression can easily accommodate, then a function can be specified instead:

```

event login( int pin ) %{
    if ( pin == atm_card.pin )
        return true;
    display( "INCORRECT PIN" );
    return false;
};

```

Precondition functions must use the C++ `return` statement explicitly to return an expression convertible to `bool`. In either form, an event need not have parameters in order to have a precondition (which could test some global data, for example). Preconditions are considered `const` member functions of the CHSM.

STATE NAMES

When referring to state names, they have the form:

```

state-name-ref:  [scope-selector] identifier-list

```

```

scope-selector:  ::
                dots

```

```

dots:           .[dots]

```

```

identifier-list: identifier [. identifier-list]

```

In the following CHSM description:

```

cluster p(q,s) is {
    cluster q(s) is {
        state s {
            alpha -> s;           // goes to q's s
            gamma -> p.s;        // goes to p's s
        }
    }
    state s;
}

```

the reference to state *s* in the transition on *alpha* goes to *q*'s child-state *s*. To go to *p*'s child-state *s*, i.e., a non-local state-name, the scope of the desired state can be specified. There are three ways to do this. The most straight-forward way is to precede the state name by that of its parent-state's name and a period, as was done in the transition on *gamma*.

In the following CHSM description, however:

```
cluster p(p,s) is {
  cluster p(q,s) is {
    cluster q(s) is {
      state s {
        alpha -> p.s; // goes to intermediate p's s
        gamma -> ::p.s; // goes to outermost p's s
      }
    }
  }
  state s;
}
state s;
```

that solution would not work due to the introduction of the new, intermediate cluster *p*. To go to the outermost *p*'s child-state *s*, precede the state name by a double-colon, as was done in the transition on *gamma*.

For both examples, the second form of referring to state-names could have been used. For example:

```
gamma -> .s; // goes to p's s
```

could have been used in the first example and:

```
gamma -> ..s; // goes to outermost p's s
```

could have been used in the second where each leading period "backs up" a scope.

To refer to a state name in global scope, it must be fully-qualified:

```
%%
chsm my_machine is {
  cluster c(s) is {
    state s { /* ... */ }
  }
}

%%
int main() {
  my_machine m;
  // ...
  if ( m.c.s.active() ) // must use fully-qualified name
    // ...
}
```

EVENT NAMES

When referring to events, they have the form:

```
event-ref:    event-name
              enter(state-name)
```

```
exit(state-name)
```

event-name: *identifier*

An *event-name* is for a user event; the others are for *enter/exit* events. *Enter/exit* events are implicitly broadcast upon the entering/exiting of states; other states can make transitions on these events like just like user events:

```
enter(s) -> t;
```

Equality

There are the global operators == and != that test whether two events are equal or not. For example:

```
alpha, beta -> s %{
    if ( event == alpha )
        // ...
};
```

or perhaps:

```
gamma, delta, epsilon -> s %{
    if ( event != gamma )
        // ...
};
```

Because there is a single instance of each event per CHSM, these functions test *identity* rather than equality; hence two events of the same name from different instances of a CHSM will not compare equal.

TRANSITION LISTS

Transition lists have the form:

```
transition-list:      transition [transition-list]
transition:          event-condition-list target-action
event-condition-list: event-condition [ , event-condition-list ]
event-condition:     event-ref [ [ condition ] ]
condition:           C++-expression
target-action:       -> target-state [ %{ C++-statement-list % } ]
                       %{ C++-statement-list % }
target-state:        state-name-ref
                       [ target-expression ]
target-expression:  C++-expression
```

Conditions

A *condition* is any valid C++ expression that yields a value convertible to `bool`. For example:

```
state s {
    alpha[ counter == 0 ] -> t;
}
```

The transition occurs only if the condition evaluates to true. An event in an *event-condition* without an explicit (user-specified) condition has an implicit (default) condition that always evaluates to true.

A transition is taken if any one of the events in an *event-condition-list* occurs and its condition, if any, is true; hence the commas can be read as “or.”

Actions

An *action* is a sequence of zero or more valid C++ statements executed only if the transition occurs. For example:

```
beta -> t %{ ++counter; %};
```

Within an action, the variable `event` is available and is a reference to the event that triggered the transition:

```
alpha, beta -> t %{
    if ( event == alpha )
        // ...
%};
```

Internal Transitions

An *internal-transition* merely performs a statement-list upon an event; no “transition” actually takes place. For example:

```
state s {
    alpha %{                // internal transition
        // ...
    %};
}
```

says that, on the occurrence of the event *alpha* (or any event derived from *alpha*), execute the C++ code in the following block, but do not exit state *s* nor transition in any way. Compare that with the “self transition” of:

```
state s {
    alpha -> s %{          // self transition
        // ...
    %};
}
```

that exits *s*, broadcasts *exit(s)*, reenters *s*, broadcasts *enter(s)*, and performs transitions as a result of said broadcasts, if any. Internal transitions can be used as an optimization where the full-blown mechanics of regular transitions are not needed. The term “internal transition” is a poor one, but it’s the term in common use.

Dominance

When a state has more than one transition with a condition (either explicit or implicit) that evaluates to true, the one declared first *dominates*:

```
state x {
    alpha -> y;           // this transition dominates...
    alpha -> z;           // ...over this one
}
```

This is more useful when the first transition has an explicit condition so it functions like an “if-else”:

```
alpha[ c ] -> y;         // if ( c == true ) ...
alpha -> z;              // else ...
```

Additionally, when a parent- and child-state both have such a transition, the parent-state's dominates:

```
cluster c(x) {
    alpha -> y;           // this transition dominates...
} is {
    state x {
        alpha -> y;       // ...over this one
    }
}
```

Note that internal transitions will *not* dominate over others on the same event:

```
cluster c(x) {
    alpha %{ /* ... */ %}; // this transition will NOT dominate...
} is {
    state x {
        alpha -> y;       // ...over this one
    }
}
state y;
```

because the internal transition doesn't really "transition," hence there is no real transition to dominate.

Target Expressions

A *target-expression* is any valid C++ expression that yields a value convertible to `CHSM::state*`. The value of the expression determines the state to transition to at run-time rather than compile-time. For example:

```
state s {
    alpha -> [ &t ];
}
state t;
```

If the expression evaluates to null, the transition is aborted. Note that returning a state that results in an illegal transition (such as a child state of a set transitioning to a sibling) results in undefined behavior.

Within a target-expression, the variable `event` is available and is a reference to the event that triggered the transition:

```
alpha, beta -> [ f( event ) ];
```

SPECIAL CONSTRUCTS

Within all C++ code for enter-exit-blocks, preconditions, conditions, and actions, the following `$` constructs can be used:

`$(state-name)`

Refers to the state *state-name*:

```
chsm my_machine is {
    cluster c(s,t) is {
        state<my_state> s {
            alpha -> t %{
                ${s}.member_function();
            }%
        }
    }
    // ...
}
```

```
    }
}
```

If the `$(state-name)` notation were not used, the fully-qualified state name would need to be used instead:

```
c.s.member_function();
```

This notation also permits the scope-resolution operators to be used inside of it.

```
$enter(state-name), $exit(state-name)
```

Refers to the enter/exit event of `state-name` (instead of the C++ library function `exit()`, for example):

```
alpha, exit(s) -> t %{
    if ( event == $exit(s) )
        // ...
%};
```

```
$in(state-name)
```

Returns true only if the CHSM is in the state `state-name`:

```
alpha[ $in(s) ] -> t;
```

The above is equivalent to:

```
alpha[ ${s}.active() ] -> t;
```

THREAD SAFETY

The CHSM specification language is “thread-safe” meaning that multiple threads can broadcast events to the same machine concurrently.

However, user-specified code in enter/exit-blocks, event preconditions, transition conditions, target expression, and actions is not thread-safe unless made so by the user.

FILES

<code>file.chsmc</code>	CHSM/C++ source file
<code>file.h</code>	intermediate C++ declaration file
<code>file.c</code>	intermediate C++ definition file

SEE ALSO

chsmc(1), **CHSM::cluster(3)**, **CHSM::event(3)**, **CHSM::machine(3)**, **CHSM::parent(3)**, **CHSM::set(3)**, **CHSM::state(3)**

Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990. pp. 178-179.

David Harel, et al. “On the Formal Semantics of Statecharts.” *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, IEEE Press, NY, 1987. pp. 54-64.

David Harel. “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming*, vol. 8, 1987. pp. 231-274.

Paul J. Lucas. “An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines.” *M.S. Thesis*, University of Illinois at Urbana-Champaign, 1993. Technical Report: UIUCDCS-R-94-1868. <http://www.paujlucas.org/resume/thesis.pdf>

AUTHORS

Paul J. Lucas <paul@lucasmail.org>

Fabio Riccardi <fabio.riccardi@mac.com>