NAME

CHSM-Java − Concurrent, Hierarchical, Finite State Machine specification language for Java

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 Java 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 Java 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 Java declarations or definitions needed to compile the resultant Java 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 Java.

Keywords

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

```
chsm   cluster   deep    enter    event   exit    final   history
in     is        param   public   set     state   upon
```

Additionally, all Java 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 Java method. All state names and all event names are in the scope of all blocks.

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 {
    // ...
}

%%
public class Example {
    public static void main( String args[] ) {
        my_machine m = new my_machine();
        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: **[**`public`**]** `chsm` *state-decl* **[**`(`*param-list*`)`**]** **[***history-decl***]** **[***machine-block***]** *parent-body*

state-decl: **[**<*class-name*>**]** *identifier*

param-list: *Java-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 `public` is given, the resulting Java class implementing the CHSM will be declared public.

If <*class-name*> is given, 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 Java function *formal* argument declarations:

```
chsm my_machine( int x ) is {
```

See **CHSM.Machine**(3) for more information.

Enter-Exit-Blocks

Enter-exit-blocks are arbitrary Java 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* `%{` *Java-statement-list* `%}`

enter-exit-selector: `enter`
`exit`

For example:

```
state s {
    upon enter %{
        System.out.println( "hello, world" );
    %}
    upon exit %{
        System.out.println( "goodbye, world" );
    %}
}
```

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 *Java-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 special `$param` construct:

```
event say( String message );

state s {
    say -> t %{
        System.out.println( $param( say, message ) );
    %};
}
```

Additionally, all parameters inherited from base events, if any, are also accessible via `$param`:

```
event<say> quote( String author );
```

```
state s {
    quote -> t %{
        System.out.println(
            $param( quote, message ) +
            "\n-- " + $param( quote, author )
        );
    %};
}
```

Preconditions

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

precondition: `[` *Java-expression* `]`
`%{` *Java-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 Java 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 Java `return` statement explicitly to return a `boolean` expression. 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 methods 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 { /* ... */ }
    }
}

%%
public class Example {
    public static void main( String args[] ) {
        my_machine m = new my_machine();
        // ...
        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

The operators operators `==` and `!=` 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 )
        // ...
%};
```

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: *Java-expression*

target-action: `->` *target-state* **[** `%{` *Java-statement-list* `%}` **]**
`%{` *Java-statement-list* `%}`

target-state: *state-name-ref*
`[` *target-expression* `]`

target-expression: *Java-expression*

Conditions

A *condition* is any valid Java `boolean` expression. 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 Java 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 Java 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 Java expression returning `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 Java 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}.method();
            %}
        }
        // ...
    }
```

```
}
```

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

```
c.s.method();
```

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 *state-name*:

```
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;
```

`$param(` *event-name*`,` *param-name* `)`
Access an event parameter *param-name*:

```
event say( String message );

state s {
    say -> t %{
        System.out.println( $param( say, message ) );
    %};
}
```

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

file`.chsmj`	CHSM/Java source file
file`.java`	intermediate Java definition file

SEE ALSO

chsmj(1), **CHSM.Cluster**(3), **CHSM.Event**(3), **CHSM.Machine**(3), **CHSM.Parent**(3), **CHSM.Set**(3), **CHSM.State**(3)

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.pauljlucas.org/resume/thesis.pdf`

AUTHORS

Paul J. Lucas <*paul@lucasmail.org*>
Fabio Riccardi <*fabio.riccardi@mac.com*>