StateText:
A Textual Language for State Charts with Data

Klaus Havelund and Kim Guldstrand Larsen

January 21, 1998

Abstract

A textual extension of VisualState with data and Hierarchy ala State Charts is provided. State Charts are adopted as defined in UML (Unified Modelling Language) while data definition and manipulation constructs are adopted from the rule based model checking language Murphi. The resulting language is a textual version of State Charts with a quite powerful data manipulation language.

Contents

1 Introduction

This document describes an extension of VisualState with data and hierarchy. Basically, we regarded our initial task as to extend with just hierarchy ala Harel's State Charts as these are defined in UML. UML, however, allows for the definition of data attributes and their occurrence in transitions: in conditions and in statements that update these attributes. We decided to define a strong attribute definition and manipulation language, and for that purpose we have incorporated essential parts of the Murphi language into our design. Murphi is a rule based model checking language with a very strong Pascal-like notation for defining and manipulating data: it allows the definition of constants, types, variables, procedures and functions, and has in addition many convenient expression and statement constructs. Hence, in a certain way, StateText can be regarded as a combination of VisualState, Murphi and UML's State Charts in a textual format.

Another way to look at it is, that UML State Charts allow text strings to occur in certain places: in attribute declarations, in transition conditions, and in transition effects (on the attributes). The idea being, that UML can be "instantiated" by plugging in a favorite programming language notation for writing these strings. We have then plugged in Murphi and VisualState.

2 Conventions

The syntax for StateText will be presented in a step-by-step manner, where syntax rules are mixed with explanatory text and examples. The syntax rules are written using the following conventions. The appendix contains the complete syntax.

- <...> : denotes nonterminal
- [...] : denotes optional
- {...} : denotes repetition zero or more times
- <a> | <b> : denotes either <a> or <b>
- (...) : denotes grouping
- %s : denotes the terminal symbol s, used for reserved symbols, i.e. %(.
- <X_list> : denotes the syntactic category defined as follows:
  <X_list> ::= <X> { , <X> }

3 The Top Level: Programs

At the top level, one defines a program, which is basically just a named decomposed state. According to State Chart terminology, decomposed states are divided into two categories: XOR-states (Exclusive OR) and AND-states, in our terminology called respectively: machines (XOR) and systems (AND). A machine is a collection of states that mutually exclude each other, while a system is a collection of states "put in parallel": the program will be in each of the states. At the top level one defines either a machine or a system:

```
<program> ::=
  program <ID> : (<machine> | <system>)

<machine> ::=
  machine
    [history[*]]
     <declaration>
  end

<system> ::=
  system
     <declaration>
  end

<declaration> ::=
  <attribute_decl> |
  <event_decl>     | <eventgroup_decl> | <action_decl>  |
  <stype_decl>     | <state_decl>      |
  <rule_decl>
```

A machine/system is defined by a collection of lexically and statically scoped declarations of respectively:

- Attributes: constants, data types, data variables, procedures and functions. These are local to the state.
- Events, event groups, and actions.
- States: be they atomic or yet decomposed into machines or systems.
- Parameterized machines/systems (`<stype_decl>`).
- Transition rules.

As an example of a program being defined as a machine consider the following:

```
program M :
  machine
    var x : 0..3;
    event e1;e2;
    state a;b;
    startstate a;
    rule e1 : a -> b
      when x < 3
      do x := x+1
    end;
    rule e2 : b -> a end;
  end
```

The State Chart corresponding to this program is shown in figure 1. A local variable (attribute) x is defined, to range over the integer interval `0..3`. The machine can be in one of two states: `a` and `b`, moved by the events `e1` and `e2`. State `a` is the startstate. The event `e1` brings the program from state `a` to state `b` under the condition that `x < 3`. If the transition is taken, the variable `x` is incremented. The event `e2` brings the program back to state `a`, unconditioned. The program will at any time be in only one of the states `a` and `b`.

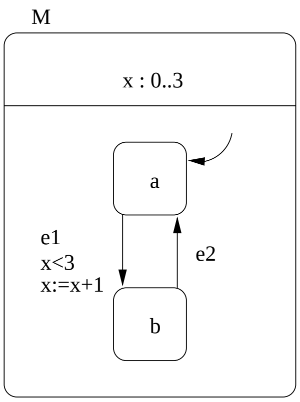

Figure 1: A machine

As an example of a program defined as a system, consider the following:

```
program S :
  system
    event e;
    state
      A : machine
            state a;b;
            rule e1 : a -> b end;
            rule e2 : b -> a end;
          end;
      B : machine
            action a;
            state p;
            rule e1 : a -> a
              action a
            end;
          end;
  end
```

The State Chart corresponding to this program is shown in figure 2. The program is the parallel composition of two states A and B, each of which is a machine. Hence, S will always be in in the composed state <a,p> or <b,p>. Note that the action a is declared local to A. In general, events, event groups and actions can be declared local to a decomposed state. It just means that the corresponding entity is only used in that state. Of course, events as well as actions are visible to the surroundings since events come from outside and actions are transmitted to the outside world. Machine B hence performs the action a whenever an e1 event occurs.

4 Attributes

As mentioned, attributes (constants, datatypes, variables, procedures and functions) can be declared locally to any machine/system. The main purpose with such declarations is to introduce a collection of local variables and a set of procedures/functions that can read and modify these variables. The variables can then be referred to in transitions, either in conditions or in update-statements. An attribute declaration is either the declaration of a constant, type or variable (<data_decl>) or the declaration of a procedure or function (<proc_decl>):

```
<attribute_decl> ::=
  <data_decl> | <proc_decl>
```

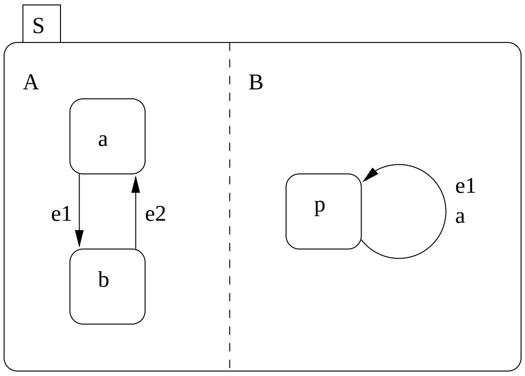

Figure 2: A system

4.1 Constants, Types and Variables

Constant, type and variable declarations are defined by the following rules:

```
<data_decl> ::=
   const { <const_def> ; }
 | type { <type_def> ; }
 | var { <var_def> ; }

<const_def> ::=
  <ID> = <expr>

<type_def> ::=
  <ID> = <typeExpr>

<typeExpr> ::=
   <ID>
 | <expr> .. <expr>
 | enum %{ <ID_list> %}
 | record { <var_def> } end
 | array %[ <typeExpr> %] of <typeExpr>

<var_def>  ::=
  <ID_list> : <typeExpr>
```

Note that all types are finite involving for example bounded integer intervals rather than general integers. The following shows some examples, and should be more or less self-explanatory.

```
const
  MAX = 5;
```

```
type
  Number = 1..MAX;
  Colour = enum{BLUE,RED};
  Data = record
           number : Number;
           colour : Colour;
         end;
  Stack = array[1..10] of Data;
var
  stack : Stack;
  next : 0..10;
```

4.2 Procedures and Functions

Procedures and function declarations are defined according to the following rules:

```
<proc_decl> ::=
   <procedure>
 | <function>

<procedure> ::=
  procedure <ID> %( [ <formal> { ; <formal> } ] %) ;
  [ { <data_decl> } begin ] [ <stmts> ] end;

<function> ::=
  function <ID> %( [ <formal> { ; <formal> } ] %) : <typeExpr>;
  [ { <data_decl> } begin ] [ <stmts> ] end;

<formal> ::=
  [var] <var_def>
```

Some self explanatory examples are:

```
procedure Add(n:Number; c:Colour);
  if next < 10 then
    next := next + 1;
    stack[next].number := n;
    stack[next].colour := c;
  end
end;

function any_blue():bool
  var found : bool;
begin
  found := false;
```

```
  for idx : 1..10 do
    if stack[idx].colour = BLUE then
      found := true
    end
  end;
  return found;
end
```

Procedures and functions can have local variables (in which case a `begin` must follow the declarations). Functions return a value by executing a `return` statement. If a formal parameter is prefixed with the keyword `var` it means the variable is called by reference.

4.3 Expressions

Expressions evaluate to values, and are defined through the following rules:

```
<expr> :=
   %( expr %)
 | <designator>
 | <integer-constant>
 | <ID> %( <actuals> %)
 | forall <quantifier> do <expr> end
 | exists <quantifier> do <expr> end
 | <expr> <bin_op> <expr>
 | not <expr>
 | <expr> ? <expr> : <expr>
 | in <state_name>

<designator> :=
  <ID> { . <ID> | %[ <expr> %] }

<quantifier> ::=
   <ID> : <typeExpr>
 | <ID> := <expr> to <expr> [ by <expr> ]

<bin_op> ::=
  + | - | * | / | % | or | and | -> | < | <= | > | >= | = | !=
```

A designator is used to refer to array and record elements, using dot-notation to access the elements of a record. It has a C-like if-expression (`e1?e2:e3` means if `e1` then `e2` else `e3`) and the normal set of binary operators plus negation. The language allows for quantified expressions over finite types, such as for example the following two expressions:

```
forall idx:1..10 do
  stack[idx].colour = RED
```

```
end

exits idx:1..10 do
  stack[idx].colour = BLUE
end
```

Finally, as is the case in VisualState, one state machine `M` can refer to the states of other machines in the sense that it can test whether another machine `N` is in a particular state `s` as follows: `in N.s`.

4.4 Statements

The kind of statements available are the classical ones, as given by the rules:

```
<stmts> ::=
  <stmt> {; [<stmt>] }

<stmt> ::=
   <assignment>
 | <ifstmt>
 | <switchstmt>
 | <forstmt>
 | <whilestmt>
 | <proccall>
 | <assertstmt>
 | <putstmt>
 | <returnstmt>
```

```
<assignment> ::=
  <designator> := <expression>

<ifstmt> ::=
  if <expr> then [ <stmts> ]
    { elsif <expr> then [ <stmts> ] }
    [ else [ <stmts> ] ]
  end

<switchstmt> ::=
  switch <expr>
    { case <expr_list> : [ <stmts> ] }
    [ else [ <stmts> ] ]
  end

<forstmt> ::=
  for <quantifier> do [stmts] end

<whilestmt> ::=
  while <expr> do [stmts] end

<proccall> ::=
  <ID> %( <expr_list> %)

<assertstmt> ::=
  assert <expr> [ <string> ]

<putstmt> ::=
  put ( <expr> | <string> )

<returnstmt> ::=
  return [ <expr> ]
```

Examples are:

```
idx := 5

stack[idx].colour := RED

if idx < 10 then
    ...
  elsif idx = 10 then
    ...
  else
  ...
end

switch colour
  case BLUE : ...
  case RED  : ...
```

```
end

for idx := 1 to 10 do ... end

while ok do ... end

F(5) -- procedure or function call

assert (idx <= 10)

put(``state x reached'')

return 4
```

The assert statement evaluates the expression and causes an error to be identified if the expression evaluates to false. The put statement outputs its arguments and can be used during simulations.

5 Events, Event Groups and Actions

Events, event groups and actions are defined by the following rules:

```
<event_decl> ::=
  event { <signal_def> ; }

<signal_def> ::=
  <ID> [ %( <typeExpr_list> %) ]

<eventgroup_decl> ::=
  eventgroup { <eventgroup_def>; }

<eventgroup_def> ::=
  <ID>(<ID_list>)

<action_decl> ::=
  action { <signal_def> ; }
```

A special observation to make here is that events as well as actions can be parameterized. Examples are:

```
type
  T = 1..4;
event
  e1; e2(1..5); e3(bool,T);
```

```
eventgroup
  e12(e1,e2);

action
  a; b(1..3);
```

The event e1 is a normal atomic event, while the events e2 and e3 are parameterized. Parameterized events occur "instantiated". That is, the following are events that can occur:

```
e1
e2(3)
e3(true,4)
```

When defining transition rules, parameterized events must occur with identifiers (formal parameters) which then are bound to the actual values, as in:

```
var
  w : 1..10;

rule e3(x,y) : a -> b
  when y < 3
  action b(y+1)
  do
    if x then
      w := y + 2
    end
end
```

The above rule fires for example if event e3(true,2) occurs. In that case, the action b(3) is fired, and the local variable w is updated to become 4 (2+2).

6 State Declarations

One of the main components of a machine/system is a collection of state declarations, following the rules:

```
<state_decl> ::=
  state { <state_def> ; }

<state_def> ::=
   <ID>
 | <ID_list> : <machineExpr>

<machineExpr> ::=
   <machine>
 | <system>
 | <instance>
```

Ignore for a moment the <instance> alternative. Examples of state declaration are then:

```
program P :
  machine
    state
      a;

      B : machine
            ...
          end;

      C : system
            ...
          end
  end
```

This declares an atomic state a, and two decomposed states A and B, the state A being a machine and the state B being a system. A <machineExpr> can be regarded as a kind of state type, and a state declaration declares an instance of this type. As a more involved example, consider the following, the State Chart of which is shown in figure 3.

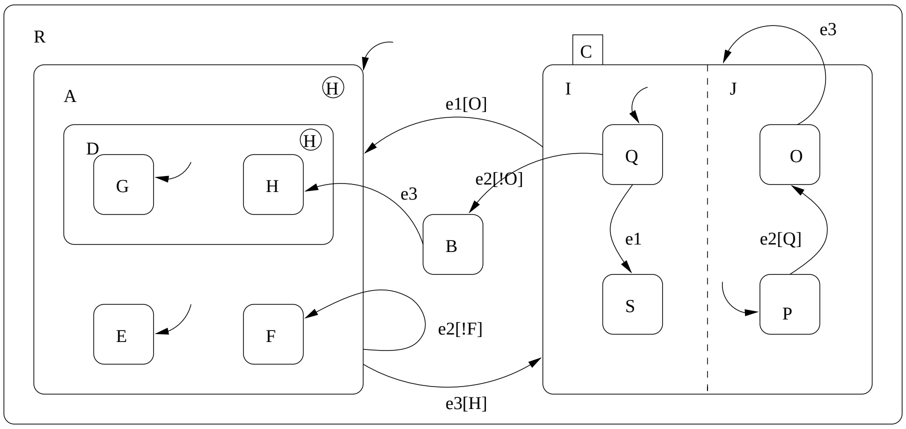

Figure 3: Nested states

```
program R :
  machine
    event e1;e2;e3;
    state
      A : machine
            history
            state
              D : machine
                    history
                    state G;H;
                    startstate G;
                  end;
              E;F;
            startstate E;
          end;

      B;

      C : system
            state
              I : machine
                    state Q;S;
                    startstate Q;
                    rule e1 : Q -> A end
                  end;
              J : machine
                    state O;P;
                    startstate P;
                    rule e2 : P -> Q when in I.Q end
                  end;
          end

    startstate A;

    rule e1 : C -> A when in C.J.O end;
    rule e3 : B -> A.D.H end;
    rule e2 : C.I.Q -> B when not in C.J.O end;
    rule e2 : A -> A.F when not in A.F end;
    rule e3 : A -> C when in A.D.H end;
    rule e3 : C.J.O -> C end;
  end
```

7 Rules

Rules define the behaviour of a machine/system within which they are defined. A rule can in conditions refer to all states within this machine and to states in all surrounding scopes. What concerns the source state and the target state of the transition, a rule can however only refer to states within the machine it is defined, including states in machines/systems nested within. The syntax for rules is as follows:

```
<rule_decl> ::=
  { <rule> ;}

<rule> ::=
   <simplerule>
 | <startstate>
 | <invariant>
```

There are three kind of rules, of which we have already seen two: simple rules and startstates. In addition to these there are invariants, which we shall explain in a moment.

7.1 Simple Rules

The syntax for simple rules is as follows.

```
<simplerule> ::=
  rule [<string>]
    [<event>] [: <state_spec> -> <state_spec>]
    [when <expr>]
    [action { <action> }]
    [do [{ <data_decl> } begin] <stmts>]
  end

<event> ::=
  <ID> [ ( <ID_list> ) ]

<action> ::=
  <ID> [ ( <expr_list> ) ]

<state_spec> ::=
   <state_name> [ %{ <state_spec_list> %} ]

<state_name> ::=
  <ID> { .<ID> } [.@]
```

The following is a quite elaborate example of the possibilities, all of which are optional:

```
var
  x : 0..10;
  y : bool;

...

rule ``a lot of stuff''
  E : a -> b
  when
    x > 0
  action
    A B
  do
    var found : bool;
  begin
    found := false;
    for idx : 1..10 do
      if stack[idx].colour = BLUE then
        found := true
      end
    end;
    y := found;
  end
```

The rule has a name, `` `a lot of stuff' `` which is a text string. This can be useful during verification when error traces are generated: an error trace will then contain a sequence of rules fired, identified by their names. The rule fires on the event `E` and goes from state `a` to state `b`. However, only under the condition that some variable `x > 0`. As a result of taking the transition, the actions `A` and `B` are fired. Finally, a variable `found` local to the rule is declared, and a statement is executed atomically, which assigns true to the global variable `y` if there is a blue element in the stack.

This is an example of a rule using all options. All of these may be omitted: conditions (`when`), actions (`action`), variable assignment (`do`) and local variables (`... begin`). Concerning the initial part of the rule:

```
E : a -> b
```

there are some conventions however: when either the event `E` or the transition `a -> b` is omitted. Some examples are shown below, which illustrate this. If the event is omitted, this can have two interpretations, as illustrated by the following two rules:

```
rule a -> b
  when Enabled()
end

rule a -> b end
```

If the rule is labelled with a condition in addition to the transition `a -> b` between states, then this rule is simply just conditioned on the corresponding condition to be true. This is allowed in UML. If on the other hand, as in the second rule, there is no condition, then the rule is supposed to be triggered whenever all final states in the source state `a` have been reached.

The second possibility is that there is no transition `a -> b`, as in the following examples:

```
rule entry
do
  Reset()
end

rule exit
do
  CleanUp()
end

rule
do
  InternalStuff()
end
```

This signifies what in UML is called "internal transitions": transitions that are taken when a state is entered (`entry`), when it is left (`exit`), and while in the state (no event is given at all). The events `entry` and `exit` are reserved keywords. The last rule states that whenever in the state where it is declared, this rule may be fired at any time and do its job. In this case the job will be atomically executed, while in UML it can in fact be a non-atomic activity.

The category of `<state_name>` needs a bit of explanation. A state name of the form:

```
A.@
```

refers to what in State Chart terminology is called a *stubbed* state. That is, some state inside `A`, but we are not aware of which, typically because state `A` is not yet decomposed. Another variant is a name of the form

```
A.final
```

This refers to the *final* state within `A`, as this is used in UML to indicate "termination" of a state. This enables any completely unlabelled transition leading away from this state.

The category `<state_spec>` also needs some explanation. Consider figure 3, which contains a system `C` which is the parallel composition of `I` and `J`. Suppose we want to specify a rule that on the event `e1` leaves `C` if this is in the states `I.S` and `J.P` and then goes to state `A`. This may be expressed as follows:

```
rule e1 : C{I.S,J.P} -> A end
```

Hence, the brackets "{" and "}" are used to indicate the occurrence of a parallel system, the name of which is `C`.

7.2 Startstates

The syntax for startstates is:

```
<startstate> ::= startstate   [ <string> ]   [ <event> :]  <ID>
  [ [ <data_decl>  begin] <stmts> end]
```

In its simplest form, a startstate declaration just show an initial state as in:

```
startstate a;
```

However, UML allows for several startstates to be given for a decomposed state, each triggered by a particular event, and in addition assignments to variables are allowed as a side effect of changing to the initial state:

```
startstate E1 : closed end;

startstate E2 : open
  window_open := true
end;
```

The above two declarations express that if the enclosing state is reached by an `E1` event, then state `closed` will be chosen, and in case of `E2` the state `open` will be chosen, in the latter case, the variable `window_open` will in addition be updated.

7.3 Invariants

Invariants are “rules” having the sole purpose of stating properties we want to hold invariantly (always). The syntax for invariants is as follows:

```
<invariant> ::=
  invariant [ <string> ] <expr>
```

Examples are:

```
invariant ''water level ok''
  water_level <= 100;

invariant ''no blue''
  forall idx:1..10 do
    stack[idx].colour = RED
  end;
```

Each of these invariants will then be checked in every state, and in case one of them is broken, an error trace will be generated showing the path from the initial state to the state where it is broken.

8 Machine Types

As can be seen from some of the former programs, such may be nested quite deeply. One way to avoid such nesting is to define and name state types, and then instantiate these at appropriate places. The general form of a statetype declaration is:

```
statetype
  M(...) =
    machine
      ...
    end
```

or

```
statetype
  S(...) =
    system
      ...
    end
```

depending on whether one defines a machine type or a system type; both are possible. As is indicated, statetypes can be parameterized, and possible parameters are: states, events, actions, and data values as for procedures and functions. This will all be clarified. Statetypes must be instantiated in state declarations with actual parameters as follows:

```
state
  m : M(...);
  s : S(...),
```

Statetype definitions are defined through the following rules:

```
<stype_decl> ::=
  statetype { <stype_def> ;}

<stype_def> ::=
  <ID>[ %( <stype_parameters> %) ] = <machineExpr>;

<stype_parameters> ::=
  <stype_parameter> { ; <stype_parameter> }

<stype_parameter> ::=
   <state_parameter>
 | <event_parameter>
 | <action_parameter>
 | <var_def>

<state_parameter> ::=
  <ID_list> : state

<event_parameter> ::=
  <signal_def_list> : event

<action_parameter> ::=
  <signal_def_list> : action

<machineExpr> ::=
   <machine>
 | <system>
 | <instance>

<instance> ::=
  <ID> [ %( <argument_list> %) ]

<argument> ::=
   <state_name>
 | <event_name>
 | <action_name>
 | <expr>

<event_name> ::= <ID>

<action_name> ::= <ID>
```

As an example, consider we want to produce the StateText code for the State Chart given in figure 4, where two quite similar alarm automata are in parallel with a socalled unit, which can be in one of two states: `safe1` and `safe2`. The two alarms are similar in the sense that they only differ in the events they react on (`set1` respectively `set2`), and in the conditions that certain transitions require in order to be fired (`safe1` respectively `safe2`).

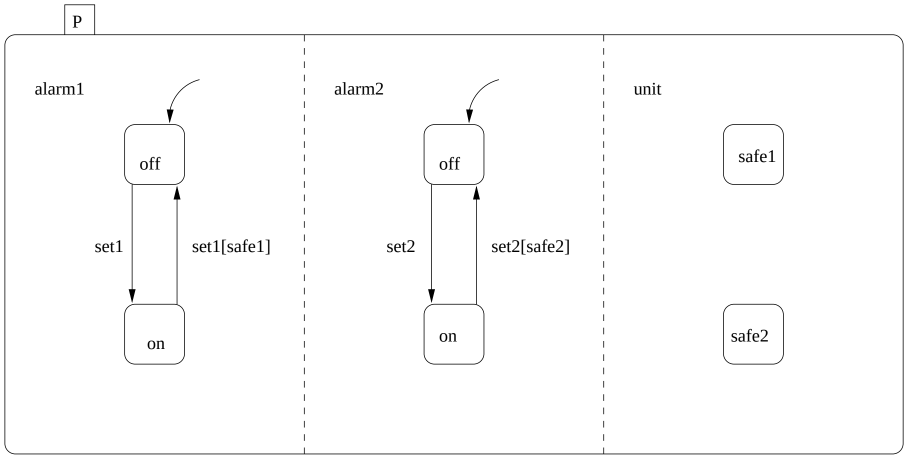

Figure 4: Two alarms of the same "type"

As a solution, we define the statetype `Alarm`, which is parameterized with the event to react on, and the state to test on. Then we instantiate this statetype in two copies, on for each alarm. This is shown below:

```
program P :
  system

    statetype
      Alarm(safe : state; change : event) =
        machine
          state off;on;
          startsate off;
          rule change : off -> on end;
          rule change : on -> off
            when in safe
          end;
        end
```

```
    event set1; set2;

    state
      unit   : machine
                 state safe1;safe2;
               end;
      alarm1 : Alarm(unit.safe1,set1);
      alarm2 : Alarm(unit.safe2,set2);

  end
```

As mentioned, statetype parameters can be of various kinds: states, events, actions and data as for procedures and functions. Some examples are:

```
type
  T = 1..5;

statetype
  M(s : state; ev(T) : event; act(T) : action; cond : bool) =
    machine
      var old : T;
      state a;b;
      rule ev(x) : a -> b
        when
          cond and
          x < 5 and
          in s
        action
          act(x+1)
        do
          old := x
      end;
    end;
```

We see how the parameterized event `ev` occurs with a formal parameter in the rule inside the machine type, the idea being that it becomes bound to the actual parameter when a particular event occurs and the rule fires.

A parameterized statetype has to be instantiated with just event names and action names, which then have to be parameterized themselves exactly the same way:

```
event
  message(T);
```

```
action
  data(T);
state
  a;
  m : M(a,message,data,true)
```

A Concrete Syntax for StateText

```
(*************)
(* Top Level *)
(*************)

<program> ::=
  program <ID> : (<machine> | <system>)

<machine> ::=
  machine
    [history[*]]
    { <declaration> }
  end

<system> ::=
  system
    { <declaration> }
  end

<declaration> ::=
  <attribute_decl> |
  <event_decl>     | <eventgroup_decl> | <action_decl>  |
  <stype_decl>     | <state_decl>      |
  <rule_decl>


(*******************)
(* <attribute_decl> *)
(*******************)

<attribute_decl> ::=
  <data_decl> | <proc_decl>

<data_decl> ::=
   const { <const_def> ; }
 | type  { <type_def> ; }
 | var   { <var_def> ; }

<const_def> ::=
  <ID> = <expr>

<type_def> ::=
  <ID> = <typeExpr>
```

```
<typeExpr> ::=
   <ID>
 | <expr> .. <expr>
 | enum %{ <ID_list> %}
 | record { <var_def> } end
 | array %[ <typeExpr> %] of <typeExpr>

<var_def>  ::=
  <ID_list> : <typeExpr>

<proc_decl> ::=
   <procedure>
 | <function>

<procedure> ::=
  procedure <ID> %( [ <formal> { ; <formal> } ] %) ;
  [ { <data_decl> } begin ] [ <stmts> ] end;

<function> ::=
  function <ID> %( [ <formal> { ; <formal> } ] %) : <typeExpr>;
  [ { <data_decl> } begin ] [ <stmts> ] end;

<formal> ::=
  [var] <var_def>

<designator> :=
  <ID> { . <ID> | %[ <expr> %] }

<expr> :=
   %( expr %)
 | <designator>
 | <integer-constant>
 | <ID> %( <actuals> %)
 | forall <quantifier> do <expr> end
 | exists <quantifier> do <expr> end
 | <expr> <bin_op> <expr>
 | ! <expr>
 | <expr> ? <expr> : <expr>
 | in <state_name>

<bin_op> ::=
  + | - | * | / | % | or | and | -> | < | <= | > | >= | = | !=
```

```
<stmts> ::=
  <stmt> {; [<stmt>] }

<stmt> ::=
   <assignment>
 | <ifstmt>
 | <switchstmt>
 | <forstmt>
 | <whilestmt>
 | <proccall>
 | <assertstmt>
 | <putstmt>
 | <returnstmt>

<assignment> ::=
  <designator> := <expression>

<ifstmt> ::=
  if <expr> then [ <stmts> ]
    { elsif <expr> then [ <stmts> ] }
    [ else [ <stmts> ] ]
  end

<switchstmt> ::=
  switch <expr>
    { case <expr_list> : [ <stmts> ] }
    [ else [ <stmts> ] ]
  end

<forstmt> ::=
  for <quantifier> do [stmts] end

<quantifier> ::=
   <ID> : <typeExpr>
 | <ID> := <expr> to <expr> [ by <expr> ]

<whilestmt> ::=
  while <expr> do [stmts] end

<proccall> ::=
  <ID> %( <expr_list> %)

<assertstmt> ::=
  assert <expr> [ <string> ]
```

```
<putstmt> ::=
  put ( <expr> | <string> )

<returnstmt> ::=
  return [ <expr> ]


(********************)
(* <event_decl>      *)
(* <eventgroup_decl> *)
(* <action_decl>     *)
(********************)


<event_decl> ::=
  event { <signal_def> ; }

<signal_def> ::=
  <ID> [ %( <typeExpr_list> %) ]

<eventgroup_decl> ::=
  eventgroup { <eventgroup_def>; }

<eventgroup_def> ::=
  <ID>(<ID_list>)

<action_decl> ::=
  action { <signal_def> ; }


(***************)
(* <stype_decl> *)
(* <state_decl> *)
(***************)

<stype_decl> ::=
  statetype { <stype_def> ;}

<stype_def> ::=
  <ID>[ %( <stype_parameters> %) ] = <machineExpr>;

<stype_parameters> ::=
  <stype_parameter> { ; <stype_parameter> }
```

```
<stype_parameter> ::=
   <state_parameter>
 | <event_parameter>
 | <action_parameter>
 | <var_def>

<state_parameter> ::=
  <ID_list> : state

<event_parameter> ::=
  <signal_def_list> : event

<action_parameter> ::=
  <signal_def_list> : action

<machineExpr> ::=
   <machine>
 | <system>
 | <instance>

<instance> ::=
  <ID> [ %( <argument_list> %) ]

<argument> ::=
   <state_name>
 | <event_name>
 | <action_name>
 | <expr>

<state_name> ::=
  <ID> { .<ID> } [.@]

<event_name> ::=
  <ID>

<action_name> ::=
  <ID>

<state_decl> ::=
  state { <state_def> ; }

<state_def> ::=
   <ID>
```

```
 | <ID_list> : <machineExpr>


(*****************)
(* <rule_decl>   *)
(*****************)

<rule_decl> ::=
  { <rule> ;}

<rule> ::=
   <simplerule>
 | <startstate>
 | <invariant>

<simplerule> ::=
  rule [<string>]
    [<event>] [: <state_spec> -> <state_spec>]
    [when <expr>]
    [action { <action> }]
    [do [{ <data_decl> } begin] <stmts>]
  end

<state_spec> ::=
   <state_name> [ %{ <state_spec_list> %} ]

<startstate> ::=
  startstate [ <string> ] [ <event> :] <ID>
    [ [ <data_decl>  begin] <stmts> end]

<invariant> ::=
  invariant [ <string> ] <expr>

<event> ::=
  <ID> [ ( <ID_list> ) ]

<action> ::=
  <ID> [ ( <expr_list> ) ]
```