Telelogic

Statemate®

Quick Reference Guide

Statemate®

Quick Reference Guide

Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF file available from **Help > List of Books**.

This edition applies to Telelogic Statemate 4.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Reserved Words and Expressions

The section provides the complete list of Statemate's reversed words and the trigger and action expressions.

Reserved Words

Reserved words are those words that cannot be used as names in Statemate because they are used by the system. If you erroneously try to use a reserved word, Statemate prevents its use in most cases. Otherwise, the error is discovered later in the process, such as during code generation or when you use the Check Model tool.

The following is a list of reserved words in Statemate.

Keyword	Description
`ac`	Abbreviation for *active.*
`active`	Possible condition or status of activity.
`all`	All elements of an array.
`and`	Logical *and*.
`any`	Any element of an array.
`break`	Exit from loop. Used in switch case statements.
`case_ada`	Case statement for Ada.
`case_c`	Case statement for C.
`ch`	Abbreviation for *changed.*
`changed`	An element's value was modified.
`dc`	Abbreviation for *deep_clear.* If you erroneously try to use this reserved word, Statemate does not catch it until later in the process.
`deep_clear`	Clears all history.
`default`	Default case.
`delay`	Delay trigger.

Keyword	Description
`dly`	Abbreviation for *delay*.
`downto`	Loop statement command.
`else`	Loop statement command.
`en`	Abbreviation for *entered.*
`end`	Loop statement command.
`entered`	Possible status of a state.
`entering`	Event generated when a state is entered; useful as a trigger for an action based on entering a state.
`entering or`	Used in trigger expressions for actions based on entering a stat.
`enum_first`	Retrieve the first enumerated value.
`enum_image`	String representation of an enumerated value.
`enum_last`	Retrieve the last enumerated value.
`enum_ordinal`	Retrieve the ordinal position of an enumerated value.
`enum_pred`	Retrieve the previous enumerated value.
`enum_succ`	Retrieve the next enumerated value.
`enum_value`	Value of an enumerated element.
`ex`	Abbreviation for *exited.*
`exited`	Event caused by exiting a state.
`exiting`	Trigger for an action based on leaving a state.
`exiting or`	Used in trigger expressions for actions based on leaving a state.
`false`	Boolean value = 0.
`fl`	Abbreviation for *q_flush*. If you erroneously try to use this reserved word, Statemate does not catch it until later in the process.
`for`	Loop statement.
`fs`	Abbreviation for *false.*
`get`	Used to have resource wait for condition.
`get!`	Abbreviation for *q_get.*
`gt!`	Abbreviation for *get*.
`hanging`	Possible condition/status of an activity.
`hc`	Abbreviation for *history_clear.* If you erroneously try to use this reserved word, Statemate does not catch it until later in the process.
`hg`	Abbreviation for *hanging.*

Keyword	Description
history_clear	Clears the history at the current hierarchical level.
if	Loop statement.
in	Possible condition of a state; condition statement.
is	Used in case statements for Ada.
length_of	Length of the specified array.
lindex	Left index value of an array.
loop	Loop statement.
make_false	Sets the given element to false.
make_true	Sets the given element to true.
N/A	Specifies that a formal parameter is not applicable to the instance.
nand	Logical *nand [not and]*.
nor	Logical *nor [not or]*.
no	Logical *not*.
ns	Abbreviation for *entering.*
null	Null.
nxor	Logical *nxor [not exclusive or]*.
or	Logical *or.*
others	Used in case statements for Ada.
peek	Abbreviation for *q_peek.*
put	Abbreviation for *q_put.*
q_flush	Clear the queue contents.
q_get	Remove the value from the front of the queue.
q_length	Return the length of the queue.
q_peek	Copy a value from the front of the queue.
q_put	Put an item on the queue.
q_urgent_put	Put an item at the beginning of the queue.
rc	Abbreviation for *receive*. **Note**: MicroC specific.
rd	Abbreviation for *read.*
read	Element has been read (event).
read_data	Action of reading an element.
receive	Message was received. **Note**: MicroC specific.

Keyword	Description
`release`	Resource was released. **Note**: MicroC specific.
`reset_element`	Reset an element to its default value
`reset_all_elements`	Reset all elements in the scope to their default values
`resume`	Resume the operation.
`return`	Identifies the output value of a function.
`rindex`	Right index value of an array.
`rl`	Abbreviation for *release*. **Note**: MicroC specific.
`rs`	Abbreviation for *resume.*
`sc`	Abbreviation for *schedule*.
`schedule`	Performs an action some time in the future.
`sd`	Abbreviation for *suspend.*
`send`	Message was sent. **Note**: MicroC specific.
`sn`	Abbreviation for *send*. **Note**: MicroC specific.
`sp`	Abbreviation for *stop*.
`st`	Abbreviation for *start/started.*
`start`	Action performed to begin activity.
`started`	Event generated when the activity becomes active.
`stop`	Action performed to halt an activity.
`stopped`	Event generated when an activity is ended.
`suspend`	Possible condition of an activity.
`switch_c`	Switch case statement for C.
`then`	Loop statement.
`timeout`	Timeout.
`tm`	Abbreviation for *timeout.*
`tmax`	Maximum operator.
`tmin`	Minimum operator.
`to`	Loop command statement.
`tr`	Abbreviation for *true.*

Keyword	Description
`true`	Boolean value = 1.
`uput`	Abbreviation for *q_urgent_put.*
`when`	Loop statement.
`when_ada`	Used in case statements for Ada.
`while`	Loop statement.
`wr`	Abbreviation for *written/write_data.*
`write_data`	Action of writing.
`written`	Element was assigned a value.
`xor`	Logical *xor [exclusive or]*.
`xs`	Abbreviation for *exiting.*

Expressions

Expressions within Statemate take the form `Trigger/Action`:

- A *trigger* expression is an event or condition that defines the criteria for a change in system status. A trigger expression can be an event expression with a guarding condition. Refer to “Trigger Expressions” for more information.
- An *action* expression specifies what to do as a consequence of a trigger occurring. Refer to “Action Expressions” for more information.

Mini-specs and static reactions can contain multiple expressions separated by double semicolons (;;).

Trigger Expressions

The following sections describe the possible trigger expressions. The topics are as follows:

- **Event Expressions**
- **Condition Expressions**

Event Expressions

A *primitive event* is one of the following:

- Named single (non array) event
- `E (K)`, the `K`'th component of an event array `E`; `K` is any integer expression

An *array of events* (also referred to as an *event array*) is one of the following:

- Named event array
- Array slice `E (K..L)`, of an event array `E`; `K` and `L` are integer expressions

Events Related to Other Elements

The following table lists the derived events that can be used as triggers within your model. A *derived event* is an event that occurs from a change in the system environment, not from any external source. Note that Statemate automatically truncates expressions. For example, if you type in `delay`, Statemate truncates it to `dly`. The table lists the truncated version of the expression.

The following operators, which are related to various types of elements, produce a single (non-array) event.

Event Expression	Occurs When	Notes
`all(E)`	All components of event array E occurred.	E is an event array.
`any(E)`	At least one component of event array E occurred.	E is an event array.
`ch(X)`	The value of X is changed.	X is data-item or condition expression or array (including array slice); can be structured or a queue.
`dly(N)`	N clock units have passed since entering the state	N is a numeric expression.
`en(S)`	State S is entered.	Used only in statecharts.
`ex(S)`	State S is exited.	Used only in statecharts.
`fs(C)`	The value of condition C is changed to false.	C is a condition expression (not an array).
`ns`	Current state is being entered.	Used only as a trigger of a reaction in state.
`rd(X)`	X is read by action `rd!`, or from a queue by `peek!` or `get!`	X is a primitive (not an alias) data-item or condition; X can be array (not a slice), array component (not a bit-array component), structured, or queue.
`sp(A)`	Activity A is stopped.	Used only in statecharts.

Event Expression	Occurs When	Notes
st	Current activity is started.	Used only as a trigger in a reactive activity.
st(A)	Activity A is started.	Used only in statecharts.
tm(E,N)	N clock units passed from the last time event E occurred.	E is event expression (not an array). N is a numeric expression.
tr(C)	The value of condition C is changed to true.	C is a condition expression (not an array).
wr(X)	X is written by action wr!, by assignment, or by put! in a queue.	X is a primitive (not an alias) data-item or condition; X can be array (not a slice), queue array component (not a bit-array component), structured, or queue.
xs	Current state is being exited.	Used only as a trigger of a reaction in state.

Compound Events

The following table lists the compound events that can be used as triggers. Operations are shown in descending order of precedence. You can use parentheses to alter the evaluation order. For example:

```
((E[C] or E2) and E3)
```

Event	Occurs When
E[C]	E occurred and condition C is true.
E1 and E2	E1 and E2 occurred simultaneously.
E1 or E2	E1 or E2, or both, occurred.
not E and [C]	E did not occur and C is true.

Predefined Events in Static Reactions and Mini-Specs

The Statemate action language supports the use of “entering” and “exiting” for static reaction triggers, and “started” for mini-spec triggers.

Examples:

```
started/ACT1;;
started or EV_1/ACT1;;
```

IN_SIM

The event expression “in_sim(ev_exp)” using the “in_sim” operator, is interpreted as “ev_exp” in simulation, and is replaced with empty_event in all other tools.

Expressions containing “in_sim(ev_exp)” can be used on transition-label, minispec and static reaction, and must appear as the first operator in trigger side of the expression.

The ELSE Trigger

You can use `ELSE` as a predefined trigger event in triggers of transitions, reactive mini-specs, and state-static reactions.

Note

- You cannot use `ELSE` as a guard on a default transition.
- When `ELSE` is used in a mini-spec or static reaction, the `ELSE` trigger is interpreted as an “else” of *all* the other triggers that exist, not just the ones that precede it in the mini-spec or static reaction.
- An `ELSE` trigger cannot be part of an expression. It must appear alone. For example, the following statement is illegal:

  ```
  else or e1
  ```

- Using two `ELSE` triggers exiting from the same source state is illegal and is reported as an error by Check Model.
- `DEFAULT` is an alias of `ELSE`.

Example:

Consider the following statement:

```
event1/action1;;else/action2;
```

When this statement is used in a static reaction, `action2` is executed if none of the other triggers in the static reaction are activated *and* the system is in-state (that is, the state is neither in “entering” nor in “exiting”).

When the statement used in a mini-spec, `action2` is executed if none of the other triggers in the mini-spec are activated and the activity is in regular operation mode or has just been started.

When the statement is used in a statechart, the `ELSE` trigger exiting from a state S1 is activated if none of the other triggers of the *compound* transitions exiting S1 are activated.

Condition Expressions

The following table lists the operators that are related to various types of elements and represent a single (non-array) condition.

Condition Expression	True When	Notes
ac(A)	Activity A is active.	Used only in statecharts.
all(C)	All components of condition C are true.	C is a condition array.
any(C)	At least one component of condition C is true.	C is a condition array.
hg(A)	Activity A is suspended.	Used only in statecharts.
in(S)	System A is in state S.	Used only in statecharts.
X1 R X2	The values of X1 and X2 satisfy the relation R. **Note:** X1 and X2 are data-item or condition expressions.	When numeric, R can be: ==, /=, >, <, <=, or >=. When strings, arrays, structured or queues, R can be ==, !=.

The following table lists the *logical operations* that use only single (non-array) conditions and represent a single condition. The operations are shown in descending order of precedence.

Condition	True When
Cl and C2	Both Cl and C2 are true.
Cl or C2	Cl or C2 or both are true.
not C	C is not true.

You can use parentheses to alter the evaluation order. For example:

```
(not((C1 or C2) and C3))
```

Note

Logical operations have lower precedence than comparison relations.

Data-Items and Data Types Used in Condition Expressions

The following operators are applicable to strings, arrays and bit-array data-items, and to user-defined types that are defined as string, array or bit-array. The result is a constant integer.

Data-Item Expression	Meaning
`length_of(A)`	Length of array, bit-array, or string A (data-item or user-defined type)
`lindex(A)`	Left index of array or bit-array A (data-item or user-defined type)
`rindex(A)`	Right index of array or bit-array A (data-item or user-defined type)

The following operator is applicable to queues:

Data-Item Expression	Meaning
`q_length(Q)`	Current number of elements in queue `Q.`

The following operators are applicable to integers and reals, and to user-defined types that are defined as integer or real.

Data-Item Expression	Meaning
`tmax`	Maximum value
`tmin`	Minimum value

The `tmin` and `tmax` operators accept one parameter, the name of the data-item or data-type, and return the defined minimum or maximum value. When the value is not defined, the operators return `OUT_OF_RANGE`.

Note the following limitations for `tmin` and `tmax`:

- You cannot use these operators on generic activity-chart or generic statechart formal parameters, or within subroutine implementations (action language, truth table, or procedural statechart).
- The analysis tools do not support dynamic evaluation of expressions with the these operators. Specifically, the following functions do not support the operators:
  - Simulation interactive expression evaluation
  - Simulation micro-step debugger
  - Sequence diagram animation
  - Generated code debugger
- Because the operators are not considered as “usage” of data, a data-item used only inside the `tmin` and `tmax` operators will not be included in the simulation scope.

Bit-Wise Operations

The following operations are relevant to integer, bit, and bit-array operands; the result is a bit-array. The list presents the operations in descending order of precedence. Parentheses can be used to alter the evaluation order. Bit-wise operations, besides the *not* operation, have lower precedence than comparison relations and numeric operations. The *not* operation has higher precedence.

A	B	A AND B	A NAND B	A OR B	A NOR B	A XOR B	A NXOR B
false	false	false	true	false	true	false	true
false	true	false	true	true	false	true	false
true	false	false	true	true	false	true	false
true	true	true	false	true	false	false	true

Note

An ampersand (for example, `A & B`) denotes concatenation

Refer to **Bit-Array Functions** for more information.

Database Conversion Operations

Database conversion operations have required and optional guidelines:

- Required conversions include the comparison operator `=` to be written as `==` and the end-of-line comment `--` to be written as `//`.
- Optional conversions are defined as synonyms, and therefore enable you to select either the old or new operator.

Database conversion operations are controlled by the following environment variables:

- `STM_CONVERT_EQ`—Changes `==` to `.EQ.`
- `STM_CONVERT_ASSIGNMENT`—Changes `:=` to `=`
- `STM_CONVERT_NE`—Changes `/=` to `!=`

To convert the operator, set the specific variable to `ON`; otherwise, no change is made.

The following table lists the revised database operators.

Old Operator	New Operator	Description	Required or Optional
`==`	`.EQ.`	Comparison operator (for special cases integer/ba/enum)	Optional
`=`	`==`	Comparison operator	Required
`:=`	`:= or =`	Assignment operator	Optional
`/=`	`/= or !=`	Not equal operator	Optional
`--`	`//`	End-of-line comment	Required

Action Expressions

Action expressions can contain multiple actions separated by semicolons (;).

The following table lists the action statements and how they appear in the language of Statemate.

Action Expression	Purpose	Notes
`dc!(S)`	Clears the history information of the descendants of state S	Used only in statecharts.
`E`	Generates the event E	E is a primitive, single event (not an array).
`fl!(Q)`	Clears queue Q	X's type is compatible with the type of the queue components. Conditional return S is optional.
`fs!(C)`	Assigns false to condition C	C is a primitive, single condition (not an array).
`get!(Q,X,S)`	Moves the head of the queue Q into data-item or condition X; returns status S	X's type is compatible with the type of the queue components.
`gt!(c)`	Waiting for resource.	"Wait Semaphore" on the condition.
`hc!(S)`	Clears the history information of state S	Used only in statecharts.
`peek!(Q,X,S)`	Copies the head of the queue Q to data-item or condition X; returns status S	X's type is compatible with type of queue components. Conditional return S is optional.
`put!(Q,X)`	Adds data-item or condition X to the tail of queue Q	X's type is compatible with the type of the queue components.
`ra!`	Resets all elements in the scope to their default values	
`rc!(DI)`	Message was received.	"Receive Message" API on a data-item.
`rd!(X)`	Reads data-item or condition X	X is a primitive (not an alias) data-item or condition, or array (including slices). Bit-array components or slices are not allowed.
`re!(EL)`	Resets element EL to its default value	
`rl!(C)`	Resource was released.	"Release Semaphore" on a condition.
`rs!(A)`	Resumes activity A	Used only in statecharts.
`sc!(K,N)`	Performs action K, delayed by N clock units	N is a numeric expression.
`sd!(A)`	Suspends activity A	Used only in statecharts.

Action Expression	Purpose	Notes
`sn! (DI)`	Message was sent.	Send message API on a data-item.
`sp!(A)`	Stops activity A	Used only in statecharts.
`st!(A)`	Activates activity A	Used only in statecharts.
`stop`	Stops the current activity	Used only in a mini-spec of a reactive activity.
`tr!(C)`	Assigns true to condition C	C is a primitive, single condition (not an array).
`uput!(Q,X)`	Adds data-item or condition X to the head of queue Q's components	X's type is compatible with the type of the queue components.
`wr!(X)`	Writes to data-item or condition X	X is a primitive (not an alias) data-item or condition, or array (including slices). Bit-array components or slices are not allowed.
X=EXP	Assigns the value of EXP to X	X is a primitive or alias data-item, array or bit-array, condition or array condition (including slices).
`X**Y`	Raises X to the Y power	

Compound, Conditional, and Iterative Actions

Action expressions can contain multiple action statements separated by a semicolon (;). The following table lists the Statemate action expressions.

Action Expression	Notes
`++`	Increment the value of the variable by 1.
`--`	Decrement the value of the variable by 1.
`AN1; AN2;`	The actions are performed concurrently. The semi-colon is optional at the end of the list.
`break`	Causes the containing loop action to terminate.
`for $I in N to \|down to L loop VAR[$I] = 0; end loop;`	`$I` is a context variable; `N` and `L` are integers.
`if C then ANI; else AN2; end if;`	`C` is a condition expression; the `else` part is optional. `AN1` and `AN2` are action expressions.
`when E then AN1; else AN2; end when;`	`E` is an event expression; the `else` part is optional. `AN1` and `AN2` are action expressions
`while C loop AN; end loop;`	`C` is a condition expression; `AN` is an action expression.

Using Variables for Look-Up Table Values

Abscissa, Ordinate, Lower Bound, and Upper Bound values can be defined as expressions using variables.

Note

Look-up table Abscissa values are not ordered by Statemate during a **Save** operation. The expressions are evaluated at run time and used in the user-defined order. Interpolation results depend on having the values in the correct order.

Functions, Operators, Switch Cases and Truth Tables

This section provides more detailed information defining functions, syntax, arguments, variables, and limitations.

Predefined Functions

A *predefined function call* has the following syntax:

```
returned-value = function(arg1,arg2,...)
```

Statemate supports the following predefined functions:

- Arithmetic Functions
- Trigonometric Functions
- Exponential Functions

Arithmetic Functions

The following table lists the arithmetic functions supported by Statemate. The table uses the following abbreviations for the argument type and return value:

- **I -** Integer
- **R -** Real
- **S -** String
- **W** - Bit-array
- **B -** Bit

Statemate converts the arguments' type when needed.

Function	Argument Type	Return Type	Meaning
`ABS(x)`	R or I	Input's type	Absolute value
`MAX(x,y)`	Mixed R and I	Input's type	Maximum value
`MIN(x,y)`	Mixed R and I	Input's type	Minimum value
`MOD(x,y)`	I1, I2	I	`I1` modulus `I2`
`ROUND(x)`	R	I	Rounded value
`TRUNC(x)`	R	I	Truncated value

Trigonometric Functions

The following table lists the trigonometric functions supported by Statemate.

Function	Argument Type	Return Type	Meaning
`ACOS(x)`	R	R	Arc cosine (in radians).
`ACOSD(x)`	R	R	Arc cosine (in degrees).
`ASIN(x)`	R	R	Arc sine (in radians).
`ASIND(x)`	R	R	Arc sine (in degrees).
`ATAN(x)`	R	R	Arc tangent (in radians).
`ATAN2(x)`	R	R	Arc tangent (in radians) with two parameters. For example, the arc tangent of `(a1/a2)`.
`ATAND(x)`	R	R	Arc tangent (in degrees).
`ATAN2D(x)`	R	R	Arc tangent (in degrees) with two parameters. For example, the arc tangent of `(a1/a2)`.
`COS(x)`	R	R	Cosine.
`COSD(x)`	R	R	Cosine (in degrees).
`COSH(x)`	R	R	Hyperbolic cosine (in radians).
`SIN(x)`	R	R	Sine.
`SIND(x)`	R	R	Sine (in degrees).

Function	Argument Type	Return Type	Meaning
SINH(x)	R	R	Hyperbolic sine (in radians).
TAN(x)	R	R	Tangent.
TAND(x)	R	R	Tangent (in degrees).
TANH(x)	R	R	Hyperbolic tangent (in radians).

Exponential Functions

The following table lists the exponential functions supported by Statemate.

Function	Argument Type	Return Type	Meaning
EXP(x)	R	R	Exponential
LOG(x)	R	R	log base e
LOG10(x)	R	R	log base 10
LOG2(x)	R	R	log base 2
SQRT(x)	R	R	Square root

Random Functions

The following table lists the random functions supported by Statemate

Function	Argument Type	Return Type	Meaning
RAND_BINOMIAL(n,p)	I, R	I	Accepts two arguments, where n>0 and 0<p<1. The returned random values are real numbers distributed according to a binomial distribution. Function: X ~ B(n,p)
RAND_EXPONENTIAL(t)	R	R	Returns random real values distributed exponentially by the value t. Use the syntax x=rand_exponential(t) to make x equal to a randomly generated number. The syntax x=random_exponential (t) is accepted, but it makes x equal to the first value in an array called random_exponential. Function: X ~ exp(t)

Function	Argument Type	Return Type	Meaning
RAND_IUNIFORM(a,b)	I, I	I	Returns random integer values distributed according to a uniform distribution in the interval [a,b]. Function: X ~ U[a,b]
RAND_NORMAL(a,b)	R, R	R	Returns random real values distributed according to a normal distribution. Function: X ~ N[a,b]
RAND_POISSON(r)	R	I	Returns random integer values distributed according to a poisson distribution. Function: X ~ P(r)
RAND_UNIFORM(a,b)	R, R	R	Returns random real values distributed according to a uniform distribution in the interval [a,b]. Function: X ~ U[a,b]
RANDOM(i)	I	R	Returns a random real value distributed uniformly between 0 and 1. If the passed argument is not 0, a new sequence of random values, whose seed is the parameter, i, is initialized. Because Statemate initiates a session with the same seed for random functions, two consecutive executions will behave identically. The advantage to this behavior is that you can reconstruct a particular execution scenario. New scenarios are produced by providing different seeds.

Bit-Array Functions

The following table lists the bit-array functions supported by Statemate.

Function	Argument Type	Return Type	Meaning
`ASHL(x,y)`	W,I	W	Arithmetic shift left by `I`, enters 0's
`ASHR(x,y)`	W,I	W	Arithmetic shift right by `I`, preserves sign
`BITS_OF(x,y,z)`	W1,I1, I2	W	Slice of bit-array expression; least significant bit of `W1` is 0. **Note**: Only supported up to 32 bits.
`EXPAND_BIT(x,y)`	B,I	W	Expand bit; creates a bit array of `I` bits, all equal `B`
`LSHL(x,y)`	W,I	W	Logical shift left by `I`, enters 0's
`LSHR(x,y)`	W, I	W	Logical shift right by `I`, enters 0's
`MUX(x,y,z)`	W1,W2, B	W	Returns `W1` if `B==0`, `W2` if `B==1`
`SIGNED(x)`	W	I	Signed value (most significant bit of `W` is a sign bit)

String Functions

The following table lists the string functions supported by Statemate.

Function	Argument Type	Return Type	Meaning
CHAR_TO_ASCII(x)	S	I	ASCII value of Ith character of S
ASCII_TO_CHAR(x)	I	S	Returns S of one character with ASCII value I
INT_TO_STRING(x)	I	S	Converts I to decimal string; I can be negative
STRING_CONCAT(x,y)	S1, S2	S	Concatenates strings
STRING_EXTRACT(x, y,z)	S, I1, I2	S	Extracts a string of length I2 from index I1 of S
STRING_INDEX (x,y,z)	S1, I, S2	I	Index of sub-string S2 within S1; –1 if not found
STRING_LENGTH(x)	S	I	String length
STRING_TO_INT(x)	S	I	Integer value of a decimal string

Note

The index of the left-most character in a string is 0.

Predefined Constants

You can use the following predefined constants:

- pi
- e

For example:

```
circumference = pi * diameter;
```

In addition, you can use the reserved word N/A in the actual binding field in the properties for a generic instance to note that a specific formal parameter is not applicable to that instance.

Combinational Assignments

A *combinational assignment* has the following syntax:

```
CE =EXP1 when COND1 else
     EXP2 when COND2 else
     . . .
     EXPN
```

In this syntax:

- **`CE` (the combinational element) -** A primitive data-item or condition, or an alias data-item
- **`EXP1` -** A data-item or condition expression
- **`COND1` -** A condition expression
- **`N` -** A number greater than or equal to 1. If `N=1`, the assignment is simply

```
CE = EXP1
```

Combinational assignments in a sequence are separated by semi-colons, like actions in a sequence.

For example:

```
DI_CE=DI_expression
DI_CE=DI-expression_1 when CO_expression
       else DI_expression_2
DI_CE=DI_expression when CO_expression_1
       else DI_expression_2 when CO_expression_2
               else DI_expression_3

CO_CE=CO_expression
CO_CE=CO_expression_1 when CO_expression
      else DI_expression_2
CO_CE=CO_expression when CO_expression_1
      else CO_expression_2 when CO_expression_2
            else CO_expression_3
```

Constant Operators and Enumerated Types

The two constant operators are as follows:

- `enum_first(T)`
  First enumerated value of `T=> T'FIRST` in Ada.

`enum_last(T)`

Last enumerated value of `T=> T'LAST` in Ada.

Parameters to these constant operators are user-defined types defined as enumerated types.

Operators Related to Enumerated Values

The following table lists the operators that support operations on enumerated values.

Operator	Ada Equivalent	Meaning
`enum_image([T’] VAL)`	`T’IMAGE`	String representation of `VAL` in `T`.
`enum_pred([T'] VAL)`	`T'PRED`	Predecessor enumerated value of `T`.
`enum_ordinal([T']VAL)`	`T'ORD`	Ordinal position of VAL in `T`.
`enum_succ([T']VAL)`	`T'SUCC`	Successor enumerated value of `T`.
`enum_value(T,I)`	`T'VAL`	Value of the i’th element in `T`.

Parameters to these operators are either enumerated values (literals) or variables. The `T'VAL` notation is used for non-unique literals. For example:

- A user-defined type `DAY` is defined as enumerated type with the following values:
  `{SUN,MON,TUE,WED,THU,FRI,SAT}`
- Another `UDT VACATION` can be defined as of type User-Type `DAY` with subrange `{FRI,SAT}`.
- Another UDT can be defined as `{SUN, MON,TUE}`.
- The order of enumerated values within the subtype should be defined as a segment of the original type. For example, `MON` must always be greater than `SUN`.

Ordinal values start with 0 (zero). The ordinal of the values of a subtype is defined by the position in the original type definition. For example:

```
enum_ordinal(DAY'FRI) == enum_ordinal(VACATION'FRI) == 5
```

Resolving Enumerated-Types Values

When multiple instances of the same Enumeration value exist in the scope, the value is resolved according to the variable type it is assigned to.

For example, assume the existence of the following in the scope:

Enumerated Data Types:

```
COLOR 1 {RED, GREEN, BLUE}
COLOR 2 {RED, GREEN, BLUE}
```

Data-Item:

```
MY_COLOR1 of type COLOR1
MY_COLOR2 of type COLOR2
```

Transition Expression:

```
[TRUE]/ MY_COLOR1=RED;MY_COLOR2=RED;
```

In this example, the first `RED` is resolved to `RED` in `COLOR1`, according to the type of `MY_COLOR`, The second `RED` is resolved to `RED` in `COLOR2`, according to the type of `MY_COLOR2`.

Limitation:

User-defined enumerated types that use a non-unique enumeration value should be defined in Global Definition Set (GDS).

Inline Operator

The inline operator can be used in the Statemate action language to insert code into the MicroC generated code.

Note

The operator is recognized only by the MicroC Code-generator. Statemate Simulation and other code generators ignore the inline operator.

Example:

On a transitions, you can write:

```
/inline('print("my text\r\n")');
```

The “printf” is written into the MicroC generated code.

Switch Cases

Switch cases are supported by Statemate for C and Ada.

C Language

The following information describes the C language switch cases in detail.

Syntax

```
switch_c <expression> { (<expression>) {
   case_c <key_value> : <actions>;[break;]
          ...
   case_c <key_value> : <actions>;[break;]
   default : <actions>;
}
```

In this syntax:

- `<expression>`—The data-item of the expression type. This can be either `Integer` or `Enumerated`.
- `<key_value>`—The value. This can be either a literal integer or an enumerated value.
- `<actions>`—The Statemate actions.

  Note: `break;` can be used as one of the actions.

Limitations

Note

- You can use non-unique, case-constant expressions; however, Statemate executes only the first one.
- Conditional breaks are not supported.
- The maximum number of case statements inside the switch statement is 256.

Translator

The Translator translates `switch/case` structures to `if/then/else` structures for simulation and code generation needs:

```
switch/case
if (expression == key_value) actions;
```

- If the `break` statement occurs in the action, control is transferred out of the `if/then/else` statement by `if (expression == key_value) actions`.
- If the `break` statement does not occur in case body, the next `if/then/else` statement expression contains the previous expression and the current expression.
- `if (expression == key_value1 || expression == key_value2) actions;`

  All default actions are concatenated as a sequence of actions and executed if all the `if/then/else` expressions are `FALSE`.

The following table shows the translation of a switch case.

Action Language	Translation

switch_c (X)
{
case_c 1:
Y++;
case_c 2:
Y=Y+2;
X++;
break;
case_c 3:
FOO1(Y);
FOO2(Y);
break;
default :
DEF_ACTION(X);
};
	if (X==1)
	{
	Y = (Y + 1);
	}
	if ((X==1)||(X==2))
	{
	Y = (Y + 2);
	X = (X + 1);
	}
	else
	{
	if (X==3)
	{
	FOO1(Y);
	FOO2(Y);
	}
	else
	{
	DEF_ACTION(X);
	}
	}

Ada Language

The following information describes the Ada language switch cases in detail.

Syntax

```
case_ada <expression> is
    when_ada <key_value> [| <key_value>] => : <actions>;
            ...
    when_ada <key_value> [| <key_value>] => : <actions>;
    when_ada others => <actions>;
end case_ada;
```

In this syntax:

- `<expression>` - The data-item (DI) of the expression type. This can be `Integer`, `Bit-Array`, or `Enumerated`.
- `<key_value>` - The value. This can be a constant literal, enumerated value, constant integer DI, or a choice list.
- `<actions>` - The Statemate actions.

  Note: `break;` cannot be used as one of the actions.

Limitations

Note the following restrictions:

- Ranges (for example, `(RED..BLUE)`) are not supported.
- The non-standard words `case_ada` and `when_ada` are used instead of `case` and `when`.
- `when_ada others` must be the last case.
- The maximum number of case statements inside the switch statement is 256.
- Remote panels are not supported.

Translator

The Translator translates `case-ada/when_ada` structures to `if/then/else` structures for simulation and code generation needs:

- A `case_ada` statement selects for execution one of a number of alternative `sequences_of_statements`; the chosen alternative is defined by the value of an expression and simply evaluated to an `if/then/else` statement. For example:

```
if (expression == key_value1) then actions;
else if(expression == key_value2) then actions;
. . .
```

- A choice list is translated as sequence of `or` statements in an `if/then/else` expression. For example, `when_ada 1| 2| 3 => <actions>` translates to:

```
if (expression == 1 || expression == 2 || expression == 3)
    then <actions>
```

- All default action concatenated as sequence of action and executed if none of the `when_ada` statements is chosen.

The following table shows the translation of a `case_ada` statement.

Action Language	Translation

`case_ada X is`
`    when_ada 1 | 2 => Y++;Y=Y+2;`
`    when_ada 3 => FOO1(Y);`
`    when_ada 4 => FOO2(Y);`
`    when_ada others => DEF_ACTION(X);`
`end case_ada`
	`if ( (X==1) || (X==2) )`
	`{`
	`   Y = (Y + 1);`
	`   Y = (Y + 2);`
	`}`
	`else`
	`{`
	`   if ( X==3 )`
	`  {`
	`      FOO1();`
	`   }`
	`   else`
	`   {`
	`      if ( X==4 )`
	`      {`
	`          FOO2();`
	`       }`
	`       else`
	`       {`
	`          DEF_ACTION();`
	`      }`
	`   }`
	`}`

Truth Tables

This section describes the format of truth tables and how they are evaluated. The topics are as follows:

- Truth Table Operators
- Special Characters
- Input Columns
- Output Columns
- Action Column
- Default Row
- Row Execution

Truth Table Operators

A value in a truth-table input column cell can be prefixed with one or more of the following operators:

<, >, <=, >=, !=, \+, ==

For example, a value of <6 in the X Input column cell causes the cell to be evaluated as TRUE only when x<6.

Special Characters

The following table lists the characters that have special meanings within truth tables.

Character	Meaning
*	Don't care
+	Event generated (input or output)
-	Event not generated (input)

Input Columns

The input columns of a truth table are similar to the following:

CO_1	CO_2	DI_1	REC_1	ARR_1
True	False	1	REC_2	{1,2,3}
False	False	2	*	*
True	False	3	*	*
False	True	5	*	*

Each column in the input section of the table is associated with an input. Inputs can be either a Statemate element or expression. Subroutine parameters and globals can be used as inputs when the truth table is a subroutine implementation body.

Compound elements can be used as inputs. For example, `CO_2` can be defined as `D1>5` and `in (STATE_1)`.

Entries in the input section can be:

- Literals
- Statemate elements
- Expressions
- Empty
- Don't care (*)

For example:

```
Row 1

CO_1 and not CO_2 and DI_1==1 and REC_1==REC_2 and
ARR_1=={1,2,3}

Row 2

not CO_1 and not CO_2 and DATA_1==2
```

Valid Input ELEMENTS

Conditions and data-items can be used as inputs to truth tables. Data-items include:

- Integers
- Reals
- Bits
- Bit-arrays
- Strings
- Records
- Record fields
- Enumerated types
- Arrays of the previously listed types
- Elements of arrays
- Subroutine calls
- User-defined types built of the previously listed types

Note

There is no literal syntax for the following types: records, unions, and arrays of complex types. The only legal comparison in the input section for these elements is another element of the same type.

Invalid Input Types

The following elements *cannot* be used as inputs:

- Unions
- Records that contain unions
- Arrays of unions
- Fields of unions
- Slices of arrays or bit-arrays
- Queues
- States
- Activities

Each input section of a row represents a Boolean expression. The Boolean expresses an AND of equivalence comparisons for each of the inputs that does not have a “Don’t Care” value.

Note

Input cells that are left blank are considered as “Don’t Care” items by the simulation and code generation tools.

Output Columns

The output columns of a truth table are similar to the following:

CO_3	DATA_2
True	100
False	-1
True	1
False	2

Each output column must be a Statemate element. Local elements, subroutine parameters, and subroutine global elements can be outputs when the truth table is a subroutine implementation body.

Entries in the cells of the output section can be:

- Literals
- Statemate elements
- Statemate expressions
- Empty

Empty entries in the output section indicate outputs that are not changed when the related row is executed. Unchanged items are not “written.”

Output Elements

Primitive conditions and data-item can be used as outputs for truth tables.

The following elements *cannot* be used as outputs:

- Compounds
- Slices of arrays
- Slices of bit-arrays
- Queues
- Activities
- States
- Actions

Note

The same element can appear in the table as both an input and an output.

Action Column

In the Action column, you can include any action expression that is legal in the context of the truth table.

The action column is similar to the following:

Action
AN1;AN2
AN3
X:=X+Y

Default Row

Optionally, you can add a default row to the truth table. This row contains no input values and is executed only if none of the previous rows in the table have been executed.

Row Execution

Statemate evaluates a truth table as follows:

- When a truth table is executed, Statemate evaluates it row-by-row, starting at the top of the table and proceeding downward to the end.
- The first row whose input expression evaluates to True is "fired."
- Once the row is fired, all the outputs listed in the output section of that row are generated and the action section is executed.
- If any output columns are blank, the related outputs are not changed. Unchanged items are not "written."
- The order of execution is from left to right—first outputs, then actions. This is relevant only for truth tables that implement procedures.
- If the table contains a default row, and if during the evaluation of the table no other row has fired, the default row is fired.
- If the table does not contain a default row and no row fires during the evaluation of the table, a warning message is displayed during simulation and no output elements are changed.

Boolean and Bit-Wise Operations on MVL Types

The following table lists NOT, AND, and OR.

IN	OUT
0	1
1	0
X	X
Z	X

IN1	IN2	OUT
0	0	0
0	1	0
0	X	0
0	Z	0
1	1	1
1	X	X
1	Z	X
X	X	X
X	Z	X
Z	Z	X

IN1	IN2	OUT
0	0	0
0	1	1
0	X	X
0	Z	X
1	1	1
1	X	1
1	Z	1
X	X	X
X	Z	X
Z	Z	X

The following table lists XOR, OP1, and OP2.

IN1	IN2	OUT
0	0	0
0	1	1
0	X	X
0	Z	X
1	1	0
1	X	X
1	Z	X
X	X	X
X	Z	X
Z	Z	X

IN1	IN2	OUT
0	0	1
0	1	0
0	X	0
0	Z	0
1	1	0
1	X	1
1	Z	0
X	X	0
X	Z	0
Z	Z	0

IN1	IN2	OUT
0	0	1
0	1	0
0	X	0
0	Z	0
1	1	1
1	X	0
1	Z	0
X	X	1
X	Z	0
Z	Z	1

Resolution Matrices

Normal	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

Wired AND	0	1	X	Z		Wired OR	0	1	X	Z
0	0	0	0	0		0	0	1	X	0
1	0	1	X	1		1	1	1	1	1
X	0	X	X	X		X	X	1	X	X
Z	0	1	X	Z		Z	0	1	X	Z

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

W

X