# DEPARTMENT OF COMPUTER SCIENCES
## UNIVERSITY OF TEXAS AT AUSTIN

*SARTOR-nnn:PC:pc*
*24 February 1990*

**TECHNICAL MEMORANDUM**

Subj: Syntax for Modechart ascii files

*Version 1.14*
*Last changed on 2/26/92*

### 1. Purpose

This note gives an overview of the syntax of the ascii files to be read and generated by Modechart. It is *not* intended as a specification; the official grammars reside elsewhere, and in case of conflict, those grammars are to be considered authoritative. It is intended to help a reader understand the basic principles governing the rules, to serve as an introduction, and to serve as a medium for review.

### 2. General rules

Input consists of a series (in any order) of definitions. A definition is one of the following:

primitive action definition
compound action definition
external event definition
state variable definition
mode definition
mode transition definition
function definition
typeclass definition

A section of this memo is devoted to each. The following rules apply to more than one section:

(1) Input is free-form, in that unlimited white space (blanks, tabs, newlines) separate tokens.

(2) Comments, which are unrestricted text delimited by `/*` and `*/`, may appear anywhere in the file. As in C, they may not nest, they may cross line boundaries, and the parser never sees them.

(3) A semicolon terminates each definition.

(4) For keywords, case is not significant; they may be given in all upper-case or all lower-case (or even mixed). For the names of user-defined objects, case is significant: "NAME" will be stored differently from "name", for example.

(5) Each definition may include a *description* just before the terminating semicolon. A description is free-form text enclosed in square brackets. Whereas the parser ignores comments, descriptions are passed on to the semantic actions, which are free to handle or ignore them as desired. Descriptions are optional and may be omitted.

(6) A syntactic unit described as a list of some sort is enclosed in { braces }, with the items separated by commas.

(7) Identifiers follow the rules for C identifiers, except that periods are allowed in identifiers as other than the first character. Keywords are reserved.

(8) Operators, where called for, follow C syntax. In particular, the assignment operator is "=", and the boolean equality operator is "==".

(9) The symbol "?" is a wild-card value that may appear in the file in place of any value token whose value is not known. It stands for "to be determined." For example, "?" may appear instead of an integer giving the maximum execution time of an action.

## 3. Using the C Preprocessor

Modechart users need the capability to define templates for mode substructures and then instantiate those templates throughout the global structure at will. To provide us with a quick implementation of this capability, we will first run the ascii file through the C preprocessor `cpp,` which gives the user a macro facility via the `#define` command.

For example, suppose that the user wished to define a canonical structure that consisted of a serial mode m1, with two children m2 and m3. Then he might include the following in his input file:

```
#define template(a) Z
SM a.m1,        CO a,   IM a.m2,        *; Z
AM a.m2,        CO a.m1,                SA a24(150,300); Z
AM a.m3,        CO a.m1,                SA a24(150,300); Z
a.m2 -> a.m3:  (alarm 50)|(deadline 25)|(1,infinity)|(delay 500);

template(X)
template(Y)
```

The first instantiation would create three modes X.m1 (child of X), X.m2 (child of X.m1), and X.m3 (child of X.m1). It would also define the transition from X.m2 to X.m3.

We can at some later date decide to process these template descriptions ourselves, so as to gain some advantage in trying to verify the resulting RTL or computation graphs. But the user won't have to know the difference, and input files that work now should work unchanged then.

## 4. Syntax for entering definitions

Each of the following subsections gives the syntax for one type of definition. Symbols in **boldface** denote terminals that must appear as shown. Symbols in *italics* denote non-terminals that will be explained later, unless their name obviates the description. Symbols in [brackets] may appear zero times or once; symbols in [brackets]$^+$ may appear 1 or more times; symbols in [brackets]$^*$ may appear zero or more times.

### 4.1. Primitive actions

A primitive action definition consists of:

*action-keyword name* **(** *positive-integer-literal* **,** *positive-integer-literal* **):**
*state-variable* **=** *function-name* **()** [ *predicate-list* ] **;**

- An *action-keyword* is **!** or **AC**.
- The *positive-integer-literal*'s denote the minimum and maximum execution times of the action, respectively.
- A *predicate-list* names predicates whose value *may* change as the result of the assignment of the function value *to this variable*. (This is opposed to predicates whose value may change simply as the result of any side-effects of executing the function, independently of the assignment of its value.)

  The predicate values may be given in one of five ways: (1) "true"; (2) "false"; (3) "?", meaning that the value is TBD; (4) "*", meaning that the value is computed nondeterministically; and (5) "<>", meaning that the value depends deterministically upon the values of the data input to the function.
- The integers denote minimum and maximum execution times.

Examples include:

```
!A1(?,?):  SYSALT = compute_altitude()  /* changes 3 predicates */
         { GT10000(?), AC_AIRBORNE(true), AC_ON_SHIP(<>) }  [description];

AC A2(?,250 V2 = turn_on()    [sets master-arm light on];

!A3(125,125):  V3 = turn_off();
```

### 4.2. Compound actions

A compound action definition consists of:

- an action keyword ( **!** or **AC**)
- a name followed by a colon
- an expression composed of parallel ("‖") and sequential (";;") operators linking other action names[1] (either compound of primitive). The operators have equal precedence, and parentheses must be used to disambiguate.
- The expression may also include synchronization points, which are denoted by <*n*>, where *n* is any unsigned nonzero integer. Synchronization points come in sets; two sync operators with the same integer tag belong to the same set. They constitute a rendezvous mechanism in the compound action definition. Each sync point is a "virtual" action that sets no variable values and takes zero time to execute, but which will suspend the action or group in which it appears until all of its set partners have been reached.

Examples include:

```
!actn: (a1 ‖ b1 ‖ c1 ) ;; (d1 ‖ (f1;;g1));
!actn: (a1 ‖ b1 ‖ c1 ) ;; (?  ‖ (f1;;?))  [description];
!actn: (a1 ‖ b1 ‖ c1 ) <1> ;; (?  ‖ <1> (f1;;?))  [description];
!actn: (a1;;a2;;a3) <2> ‖ (b1 ;; <2> b2 ;; b3)
!act13:  g2 ;; /* comment */  h2;
```

### 4.3. External events

An external event definition consists of the event keyword **ˆ** or **EV** followed by the name of the event. An optional nonnegative integer literal specifies the minimum separation between two consecutive occurrences of the event. Examples include:

```
ˆ  E1   [AC takes off] ;
EV E2 10  [aircraft lands] ;  /* E1 and E2 can't happen simultaneously */
```

### 4.4. State variables

A state variable definition is a C-like declaration consisting of a type name[1], a variable name, and an initial value assignment. The initial value may be "?" (unknown) or "*" (don't care).

The parser recognizes real, integer, boolean, and string literals. Boolean literals are "true" and "false". A string literal is a double-quoted string; string literals may extend across line boundaries, and may contain escaped double quotes. The parser does no type-checking. That is left to the semantic actions, if desired. Under development right now are the rules to recognize expressions.

Examples include:

```
real  SYSALT = 100.756 [units = feet]; /* init value is altitude of base */
integer     iv1 = -145;
integer     iv2 = 0;
integer     iv3 = +145;
boolean     b1 = false;
boolean     b2 = true;
othertype   o1 = ?;
othertype   o2 = *;
altitude    ALT = "best system estimate of current altitude" [rules TBD] ;
```

### 4.5. Function definitions

A function definition defines a function that defines a primitive actions. It consists of:

- a name followed by "()" followed by a colon
- a string literal that defines its body. This might be C code, but the parser does not examine it.
- an optional list of predicate names[1], each given with a boolean value. These are the predicates that might change value as the result of evaluating the function. As in the predicate list for primitive actions, the boolean values may be "?", "<>", or "*".

Examples include:

```
f1():  "   c code defining the parameterless function  " { p1(*) };
f2():  "   c code defining the parameterless function  " { p2(<>),p3(true) };
f3():  "   c code defining the parameterless function  " [description] ;
```

### 4.6. Mode transitions

A mode transition consists of:

- the source and destination mode names[1] separated by **->**, followed by a colon
- the transition expression. The transition expression is a set of disjuncts separated by "|". Each disjunct is either a trigger or a lb-ub pair. A trigger is a set of conjuncts, separated by "&". Parentheses may be used to group. The different kinds of conjuncts are described in Table 1.

| Conjunct form | Event or condition | Interpretation |
|---|---|---|
| ˆ*name* | E | *name* is interpreted as the name of an external event. |
| \|-*name* | E | start action: *name* is interpreted as the name of an action. |
| -\|*name* | E | stop action: *name* is interpreted as the name of an action. |
| *name* =true | E | *name* is interpreted as the name of a boolean state variable; this event refers to the occurrence of the variable being assigned the value "true". |
| *name* =false | E | *name* is interpreted as the name of a boolean state variable; this event refers to the occurrence of the variable being assigned the value "false". |
| ->*name* | E | *name* is interpreted as the name of a mode; this event refers to the occurrence of the mode being entered. |
| *name* -> | E | *name* is interpreted as the name of a mode; this event refers to the occurrence of the mode being exited. |
| *name* 1->*name* 2 | E | *name* 1 and *name* 2 are interpreted as names of modes; this event refers to the transition from *name* 1 to *name* 2 being taken. |
| *name* ==true | C | *name* is interpreted as the name of a mode; this condition is satisfied if and only if the system is currently in the mode. |
| *name* ==false | C | *name* is interpreted as the name of a mode; this condition is satisfied if and only if the system is not currently in the mode. |
| {(m1,m2,...,mn)} | C | |
| {(m1,m2,...,mn>} | C | |
| {(m1,m2,...,mn]} | C | |
| {<m1,m2,...,mn)} | C | |
| {<m1,m2,...,mn>} | C | a comma-separated list of mode names[1] enclosed in braces. |
| {<m1,m2,...,mn]} | C | |
| {[m1,m2,...,mn)} | C | |
| {[m1,m2,...,mn>} | C | |
| {[m1,m2,...,mn]} | C | |
| *name* | C | *name* is interpreted as the name of a boolean state variable. The condition is true if and only if the variable is true. |
| !*name* or !(*name* ) | C | *name* is interpreted as the name of a boolean state variable. The condition is true if and only if the variable is false. |

**Table 1**
**Forms of conjuncts in a trigger disjunct of a mode transition expression**

An lb-ub pair is one of:

- an ordered pair of non-negative integer literals
- (**alarm** *n*) where *n* is a non-negative integer literal
- (**deadline** *n*) where *n* is a non-negative integer literal or **infinity**;
- (d**elay** *n*) where *n* is a non-negative integer literal
- (*n* , **infinity**) where *n* is a non-negative integer literal

Examples of transition definitions include:

```
m1->m2:   {(m1,m2,m3)} | ({<m4]})&-|c | |-b&-|c | (alarm 1) | (20,3);
m3->m4:   (alarm 50) | (deadline 25) | (1 infinity) | (delay 500) ;
m4->m5:   ->m1 & s2=false & m3->m4 & pred1=true & !pred2 & ^ee1;
m5->m6:   (ac_airborne == false & on_carrier == true) | {<m4>}
```

### 4.7. Modes

A mode definition consists of:

- a specification of whether the mode is serial ("SM"), parallel ("PM"), or atomic ("AM);
- the name of the mode;
- the name of the mode that this mode is the Child Of ("CO" followed by the name[1] of the parent mode); the parent mode may be given by "*" if this mode is a root;
- if serial, the name of the initial mode ("IM" followed by a mode name[1]);
- a specification of the native action associated with the mode. If there is none, this is given by "*". Otherwise, a periodic action ("PA") or sporadic action ("SA") keyword, followed by the name of the action[1], followed by an ordered pair of positive integer literals. The first one is taken to be the deadline; the second is the period if the action is periodic, and the minimum separation otherwise.

Examples:

```
PM m1, CO M2,         PA a21(/* deadline */ 150, /* period */300);
AM m1, CO M2,         PA a21(/* deadline */ 150, /* period */300);
SM m3, CO *,  IM I1,  SA a23(/* deadline */ ?  , /* min. sepn. */ +50);
SM m3, CO ?,  IM *,   SA a23(1, 10);
SM m3, CO *,  IM ?,   SA a23(1, ?);
SM m3, CO ?,  IM ?,   SA a23(1, 999);
```

### 4.8. Typeclasses

The user may define new typeclasses in the Modechart file to which his state variables belong. We distinguish two kinds: numeric and enumerated. An enumerated typeclass is one whose variables may take on a finite and enumerated set of values, and for which the normal mathematical operations are not defined.

A typeclass definition consists of:

- a keyword **ETC** (for enumerated) or **NTC** (for numeric);
- the name of the typeclass;

---

[1] The parser does not check to see that the identifiers given actually name the desired type of object. The parse will succeed if any legal identifier is given. It is up to the semantic actions to check what kind of object, if any at all, the identifier refers to.

- for numerics, an optional list of physical units.
- for enumerated typeclasses, a non-optional list of values in the domain.

For example:

```
NTC real; /* no units needed for real */
NTC distance {feet, nmi, km, inches};
NTC altitude {feet} [min=-1000, max=50000];

ETC boolean {true, false};
ETC master_arm_switch {boc, bocoffset, norm_attack, norm_offset, off};
```

## 5. Design issues

(1) We felt it was important not to require that mode names be unique. We can disambiguate by referring to the modes' ancestors up to the level necessary to avoid confusion. How do we adjust the syntax to allow for this?

There are two approaches. The first is to introduce scope delimiters, and force the user to define all modes under a particular parent at the same time. The second is to force the user to write fully qualified mode names (e.g., "grandparent.parent.child1") to disambiguate. The tradeoff is long, awkward names versus the freedom to organize the input in any order.

Happily, the `#define` facility gives us both alternatives for free. The user is always allowed to write down qualified names to resolve ambiguities. However, he can also use the `#define` facility to define and re-define a short text macro to stand for various long prefixes at different points in his text file, thus achieving the effect of scope delimiters.

## 6. Summary example

The individual examples did not always appear very readable, because they were designed to show the widest variety of possible forms. Attached is an example of a complete input file, showing possible format conventions. The file was printed using the UNIX utility *vgrind,* with suitably modified definitions for Modechart input files.

**Paul C. Clements**