Appendix

A

Names and Expressions

This appendix presents the syntax rules for names and expressions in the languages described in the book.

A.1 Names

A.1.1 Reserved words

```
ac active all and any
break
ch changed
dc deep_clear downto
else en end entered entering ex exited exiting
false for fs fl
get
hanging hc hg history_clear
if in
length_of lindex loop
make_false make_true
nand nor not ns nxor
or
put peek
q_put q_urgent_put q_get q_peek q_flush q_length
rd read read_data resume rindex rs
schedule sd sp st start started stop stopped
suspend
then timeout tm to tr true
uput
wr write_data written when while
xor xs
```

A.1.2 Textual element names

- A legal name of a textual element is a sequence of alphanumeric characters, excluding blanks, and possibly including _ (underscore). It must begin with a letter.
- The maximal length of a name is 31 characters.

- Names are not case-sensitive.
- Synonyms contain at most 16 characters.
- A name cannot be a reserved word.
- A name cannot be the same as the name of a predefined function.
- User-defined types cannot have the following names: `integer`, `real`, `bit`, `array`, `queue`, `record`, `union`, `bit_array`, `string`, `condition`, `single`.
- When referring to a textual element in an expression (e.g., in a transition label), names can be spread out over multiple lines and \ (back slash) must be written before the "new-line" inside the name.
- A textual element can be referred to outside the model prefixed by the chart name in which it is defined: `chart-name:element-name` (e.g., `MAIN:X`).

A.1.3 Box element names

- A legal name of a box element is a sequence of alphanumeric characters, excluding blanks, and possibly including _ (underscore). It must begin with a letter.
- The maximal length of a name is 31 characters.
- The names are not case-sensitive.
- Synonyms contain at most 16 characters.
- The name cannot be a reserved word.
- A box element can be referred to by pathname, i.e., preceded by its parents's name: `...grandparent-name.parent-name.box-name` (e.g., `A.B.C`) and optionally also with the chart-name in which it is defined: `chart-name:pathname` (e.g., `MAIN:A.B`).
- The pathname of a top level box is: `.box-name` (e.g., `.TOP`)
- When referring to a box element in an expression (e.g., in a transition label), names can be spread out over multiple lines, and \ (back slash) must be written before the "new-line" inside the name.

A.1.4 Names of elements in generic instances

- An element in a generic instance is referred to by: `instance-name^unique-element-name-in-instance.`
- An instance name can have several levels of nesting (instance in instance in instance, etc.), in which case, several ^ signs are used.
- An instance name (box name) on each level of the nesting and the element name in the instance must be unique. Therefore, each may contain a chart name. For example, `A:K^L^B:M^C:X.`

A.2 Expressions

A.2.1 Event Expressions

Atomic event and array of events An *atomic event* is one of the following:

- Named single (nonarray) 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 operators, which are related to various types of elements, produce a single (nonarray) event.

Event	Abbreviation	Occurs when	Note
`entered(S)`	`en(S)`	State `S` is entered	Used only in statecharts
`exited(S)`	`ex(S)`	State `S` is exited	Used only in statecharts
`entering`	`ns`	Current state is being entered	Used only as trigger of reaction in state
`exiting`	`xs`	Current state is being exited	Used only as trigger of reaction in state
`started(A)`	`st(A)`	Activity `A` is started	Used only in statecharts
`started`	`st`	Current activity is started	Used only as trigger in reactive activity
`stopped(A)`	`sp(A)`	Activity `A` is stopped	Used only in statecharts
`changed(X)`	`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
`true(C)`	`tr(C)`	The value of condition `C` is changed to true	`C` is condition expression (not array)
`false(C)`	`fs(C)`	The value of condition `C` is changed to false	`C` is condition expression (not array)
`read(X)`	`rd(X)`	`X` is read by action `rd!`, or from a queue, by `peak!` or `get!`	`X` is primitive (not alias) data-item or condition; `X` can be array (not slice), array component (not bit-array component), structured and queue

Event	Abbreviation	Occurs when	Note
`written(X)`	`wr(X)`	`X` is written by action `wr!`, by assignment, or by `put!` in queue	`X` is primitive (not alias) data-item or condition; `X` can be array (not slice), array component (not bit-array component), structured or queue
`timeout(E,N)`	`tm(E,N)`	`N` clock units passed from last time event `E` occurred	`E` is event expression (not array); `N` is numeric expression
`all(E)`		All components of event array `E` occurred	`E` is event array
`any(E)`		At least one component of event array `E` occurred	`E` is event array

Compound events

The following operations use only single (nonarray) events and conditions and produce a single event.

Event	Occurs when
`E[C]`	`E` occurred and the condition `C` is true
`[C]`	Condition `C` is true
`not E`	`E` did not occur
`E1 and E2`	`E1` and `E2` occurred simultaneously
`E1 or E2`	`E1`, `E2`, or both, occurred

The list presenting operations is in descending order of precedence. Parentheses can be used to alter the evaluation order.

A.2.2 Condition expressions

Atomic condition and array of conditions An *atomic condition* is one of the following:

- Literal constant: `true`, `false` (not case-sensitive).
- Named single (nonarray) condition (can be of user-defined type).
- `C(K)`, the *k*th component of a condition "indexable" array `C`; `K` is any integer expression.
- `R.C`, a field expression of type condition in a record or union `R`, for example, `A.B.C`, where `C` is a field of type condition in the field `B` (with a record structure) in the record `A`.

An *array of conditions* (also referred to as *condition array*) is one of the following:

- Literal constant: `{C1,C2,...,K*CN,...,*CL}`; each `Ci` is a literal constant condition, and `K` is a literal constant integer.
- Named condition array (can be of user-defined type).
- `R.C`, a field expression in a record or union of a type condition array.
- Array slice, `C(K..L)`, of a condition indexable array `C` (defined next); `K` and `L` are integer expressions.

An *indexable condition array* is one of the following:

- Named condition array (can be of user-defined type)
- `R.C`, a field expression in a record or union of a type condition array.
- A component of an array, whose type is a condition array, for example: `RRC(I)`, where `RRC` is an array of condition arrays. `RRC(I)` is an array of conditions, and `RRC(I)(K)` is a condition.

Conditions related to other elements The following operators, which are related to various types of elements, produce a single (nonarray) condition.

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

Compound conditions The following *logical operations* use only single (nonarray) conditions and produce a single condition.

Condition	True when
`not C`	`C` is not true
`C1 and C2`	Both `C1` and `C2` are true
`C1 or C2`	`C1` or `C2` or both are true

The list presents the operations in descending order of precedence. Parentheses can be used to alter the evaluation order.

Logical operations have lower precedence than comparison relations.

A.2.3 Data-item expressions

Data-item expressions are converted to the required type when needed:

- Bit-arrays shorter than 32 bits to integer and vice versa.
- Bit to integer.
- Integer to real.

Therefore, *integer expression* means also expression of type bit and bit-array (with length less than 32); *numeric expression* means real expression and integer expression, including bit-array expressions (with length less than 32).

Atomic, array, and structured data-items An *atomic numeric data-item* is one of the following:

- Literal constant:
  integer: decimal integer (of value less than 2**31)
  bit-array: `0X...`(hexadecimal); `0B...` (binary); `0O...` (octal)
  real: `dec.dec[(E|e)[+|-] dec]` (`dec`=decimal integer).
- Named real, integer bit-array, or bit (can be of user-defined type).
- Named data-item defined as numeric expression.
- `D(K)`, the *k*th component of a numeric indexable array or bit-array `D`, where `K` is any integer expression.
- `R.C`, a field expression in a record or union of numeric type. For example: `A.B.C`, where `C` is a field of numeric type in the field `B` (whose type is record), in the record `A`.

An *atomic string data-item* is one of the following:

- Constant literal: sequence of characters enclosed by single quotation marks (e.g. `ABC´); maximal length is 79 characters.
- Named string (can be of user-defined type).
- Named data-item defined as string expression.
- S(K), the *k*th component of a string indexable array S, where K is any integer expression.
- R.C, a field expression in record/union of string type.

An *array of data-items* is one of the following:

- Literal constant: {D1,D2,...,K*DN,...,*DL}, where each Di is a numeric or string literal constant data-item, and K is a literal constant integer.
- Named bit-array, array of any type, or user-defined array type.
- R.D, a field expression in a record/union, whose type is a data-item, array, or bit-array.
- Array slice, D(K..L), of an indexable data-item array or bit-array D, where K and L are integer expressions.
- A component of an array, whose type is a data-item, array, or bit-array.
- Named data-item defined as an array or bit-array expression.

An *"indexable" data-item array* is one of the following:

- Named bit-array, array of any type, or user-defined array type.
- R.D, a field expression in a record/union, whose type is a data-item, array, or bit-array.
- A component of an array, whose type is a data-item, array, or bit-array.

A *structured data-item, record,* or *union,* is one of the following:

- Named data-item defined as record or union (can be a structured user-defined type).
- R.S, a field expression in a record/union of a type structured data-item.
- A component of an array, whose type is a structured data-item.

Queue data-items are data-items, array components, or record or union fields defined in the Data Dictionary as having the structure queue (directly or via a user-defined type).

Data-items related to other elements 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.

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

The following operator is applicable to queues:

Operator	Meaning
`q_length(Q)`	Current number of elements in queue `Q`

Compound data-item expressions

Numeric operations. The following operations are relevant to integer, bit, bit-arrays (of length less than 32), and real operands; the result is numeric:

```
+EXP, -EXP
EXP1**EXP2
EXP1*EXP2, EXP1/EXP2
EXP1+EXP2, EXP1-EXP2
```

The list presents the operations in descending order of precedence. Parentheses can be used to alter the evaluation order.

Numeric operations have higher precedence than comparison relations and logical operations.

Bitwise operations. The following operations are relevant to integer, bit, and bit-array operands; the result is a bit-array:

```
not EXP1
EXP1 & EXP2 (denotes concatenation)
EXP1 and EXP2, EXP1 nand EXP2
EXP1 or EXP2, EXP1 nor EXP2
EXP1 xor EXP2, EXP1 nxor EXP2
```

The list presents the operations in descending order of precedence. Parentheses can be used to alter the evaluation order.

Bitwise operations other than the `not` operation have lower precedence than comparison relations and numeric operations. The `not` operation has higher precedence.

A.2.4 Action expressions

Actions manipulating other elements

Action	Abbreviation	Does	Note
E		Generates the event E	E is primitive single event (not array)
make_true(C)	tr!(C)	Assigns true to condition C	C is primitive single condition (not array)
make_false(C)	fs!(C)	Assigns false to condition C	C is primitive single condition (not array)
X:=EXP		Assigns the value of EXP to X	X is primitive or alias data-item, array or bit-array, condition or array condition (including slices)
start(A)	st!(A)	Activates activity A	Used only in statecharts
stop(A)	sp!(A)	Stops activity A	Used only in statecharts
stop		Stops the current activity	Used only in mini-spec of reactive activity
suspend(A)	sd!(A)	Suspends activity A	Used only in statecharts
resume(A)	rs!(A)	Resumes activity A	Used only in statecharts
read_data(X)	rd!(X)	Reads data-item or condition X	X is primitive (not alias) data-item or condition, or array (including slices); bit-array components or slices are not allowed
write_data(X)	wr!(X)	Writes to data-item or condition X	X is primitive (not alias) data-item or condition, or array (including slices); bit-array components or slices are not allowed
history_clear(S)	hc!(S)	Forgets history information of stat S	Used only in statecharts
deep_clear(S)	dc!(S)	Forgets history information of descendants of state S	Used only in statecharts
schedule(K,N)	sc!(K,N)	Performs action K delayed by N clock units	N is numeric expression
q_put(Q,X)	put!	Adds data-item or condition X to tail of queue Q	X's type is compatible with type of queue components

(*Continued*)

Actions manipulating other elements (*Continued*).

Action	Abbreviation	Does	Note
`q_urgent_put(Q,X)`	`uput!`	Adds data-item or condition `X` to head of queue `Q`	`X`'s type is compatible with type of queue components
`q_get(Q,X,S)`	`get!`	Moves head of the queue `Q` into data-item or condition `X`; returns status `S`	`X`'s type is compatible with type of queue components; condition `S` is optional
`q_peek(Q,X,S)`	`peek!`	Copies head of queue `Q` to data-item or condition `X`; returns status `S`	`X`'s type is compatible with type of queue components; condition `S` is optional
`q_flush(Q)`	`fl!`	Clears queue `Q`	

Compound, conditional, and iterative actions Action expressions may contain context variables: `$legal-name`, of no more than 16 characters (see Sec. A.1). Context variables are allowed for any type of data-item or condition.

Action expression	Note
`AN1;AN2`	The actions are performed sequentially; the `;` is optional at the end of the list
`if C then AN1 else AN2 end if`	`C` is a condition expression; the `else` part is optional
`when E then AN1 else AN2 end when`	`E` is an event expression; the `else` part is optional
`for $I in K to|downto L loop AN end loop`	`$I` is a context variable; `K` and `L` are integer expressions; `AN` is an action expression
`while C loop AN end loop`	`C` is a condition expression; `AN` is an action expression
`break`	Causes the containing loop action to terminate

A.2.5 Data-type expressions

Data-types of a record or union's fields can be defined (textually) in the Data Dictionary entry of the record or union using the following syntax. Note that fields of a structured type (record and union) cannot be defined directly but only via user-defined types.

The keywords and the element identifiers are not case-sensitive. N below is a constant integer expression, that is, literal integer constant, named integer constant, or operation returning a constant value.

Square brackets denote an optional segment.

Basic types

```
integer
integer length=N
integer min=N1 max=N2
real
string [length=N]
bit
bit-array [N1 to N2]
condition
<user-defined type> (identifier)
```

Compound types

```
array [N1 to N2] [of <basic type>]
queue [of <basic type>]
```

A.3 Predefined Functions

A *predefined function call* has the following syntax:

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

To describe the arguments's type and the returned value in the following table, we use the following abbreviations: I=Integer, R=Real, S=String, W=Bit-array, B=Bit.

Conversion of the arguments's type is carried out when needed.

A.3.1 Arithmetic functions

Function	Arguments	Returns	Meaning
MAX	Mixed R and I	Input's type	Maximum value
MIN	Mixed R and I	Input's type	Minimum value
TRUNC	R	I	Truncated value
ROUND	R	I	Rounded value
ABS	I or R	Input's type	Absolute value
MOD	I1, I2	I	I1 modulus I2

A.3.2 Trigonometric functions

Function	Arguments	Returns	Meaning
SIN	R	R	Trigonometric sine
COS	R	R	Trigonometric cosine
TAN	R	R	Trigonometric tangent

A.3.3 Random functions

Function	Arguments	Returns	Meaning
RAND_EXPONENTIAL	R	R	Random exponential
RAND_BINOMIAL	I, R	I	Random binomial
RAND_POISSON	R	I	Random poisson
RAND_UNIFORM	R, R	R	Random uniform
RAND_IUNIFORM	I, I	I	Random integer uniform
RAND_NORMAL	R, R	R	Random normal
RANDOM	I	R	Random

A.3.4 Bit-array functions

Function	Arguments	Returns	Meaning
SIGNED	W	I	Signed value (m.s.b. of W is a sign bit)
ASHL	W, I	W	Arithmetic shift left by I, enters 0s
ASHR	W, I	W	Arithmetic shift right by I, preserves sign
LSHL	W, I	W	Logical shift left by I, enters 0s
LSHR	W, I	W	Logical shift right by I, enters 0s
BITS_OF	W1, I1, I2	W	Slice of bit-array expression; l.s.b of W1 is 0
EXPAND_BIT	B, I	W	Expand bit; creates a bit array of I bits, all equal B
MUX	W1, W2, B	W	Returns: W1 if B=0, W2 if B=1

A.3.5 String functions

Note: the index of the leftmost character in a string is 0.

Function	Arguments	Returns	Meaning
STRING_EXTRACT	S, I1, I2	S	Extracts a string of length I2 from index I1 of S
STRING_INDEX	S1, I, S2	I	Index of substring S2 within S1; −1 if not found
STRING_CONCAT	S1, S2	S	Concatenates strings
STRING_LENGTH	S	I	String length
CHAR_TO_ASCII	S	I	ASCII value of first character of S
ASCII_TO_CHAR	I	S	Returns S of one character with ASCII value I
INT_TO_STRING	I	S	Converts I to decimal string; I can be negative
STRING_TO_INT	S	I	Integer value of a decimal string

A.4 Reactions and Behavior of Activities

A.4.1 Statechart labels

A *statechart label* is one of the following:

- `trigger`, which is a single event expression; note that `[condition]` is a legal event expression.
- `reaction`, which is of the form `trigger/action`.
- `/action`.

A.4.2 State reactions and reactive mini-specs

A state reaction and a reactive mini-spec is a list of one or more reactions (i.e., of the form `trigger/action`) separated by `;;`:

```
reaction;; reaction;;
reaction;;
...
```

The `;;` is optional at the end of the list.

Restrictions on events, conditions, and actions depend on whether they are used in a state or activity. See Sec. A.2.

A.4.3 Procedure-like mini-spec

A *procedure-like mini-spec* has the syntax of an action. See Sec. A.2.4.

A.4.4 Combinational assignments

A *combinational assignment* has the following syntax:

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

Here, `CE` (the *combinational element*) is a primitive data-item or condition, or it is an alias data-item. `EXPi` is a data-item or condition expression. `CONDi` is a condition expression. `N` can be equal to 1 (in which case the assignment is just `CE:=EXP1`) or more.

Combinational assignments in a sequence are separated by `;`, like actions in a sequence.

A.5 Flow of Information

A.5.1 Flow labels and information-flow components

Flow labels in activity-charts and module-charts and information-flow components can be any primitive (variable) data element (event,

condition, data-item) or information flow. In addition they can be components on any level of a primitive data element (array component, array slice, and record or union field). Array components can use only literal constants.

A.5.2 Actual bindings of generic parameters

Actual bindings of parameters in generic instances have the same syntax as flow labels. See Sec. A.5.1.

Appendix

B

Early Warning System Example

Functional Decomposition Approach

B.1 Textual Description of the System

The early warning system (EWS) receives a signal from an external sensor. When the sensor is connected, the EWS processes the signal and checks whether the resulting value is within a specified range. If the value of the processed signal is out of range, the system issues a warning message on the operator display and posts an alarm. If the operator does not respond to this warning within a given time interval, the system prints a fault message on a printing facility and stops monitoring the signal. The range limits are set by the operator. The system becomes ready to start monitoring the signal only after the range limits are set. The limits can be redefined after an out-of-range situation has been detected or after the operator has deliberately stopped the monitoring. See Fig. 1.1.

B.2 The Model

B.2.1 The hierarchy of charts

Figure B.1 depicts the hierarchy of charts in the EWS model.

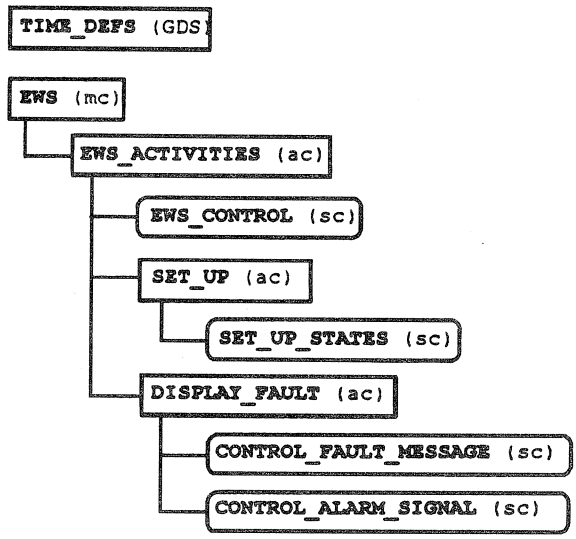

Figure B.1 Hierarchy of charts in the EWS model.

B.2.2 The charts

Figures B.2–B.8 depict the charts in the EWS model.

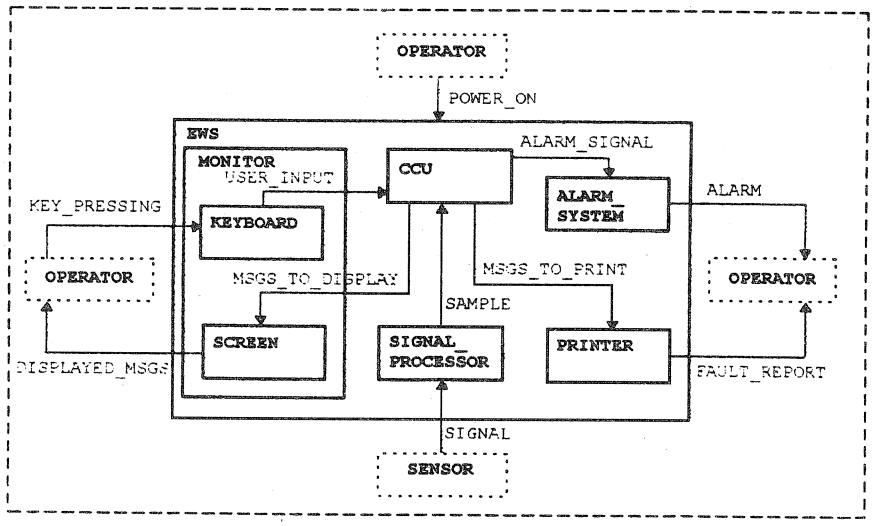

Figure B.2 Module-chart EWS model.

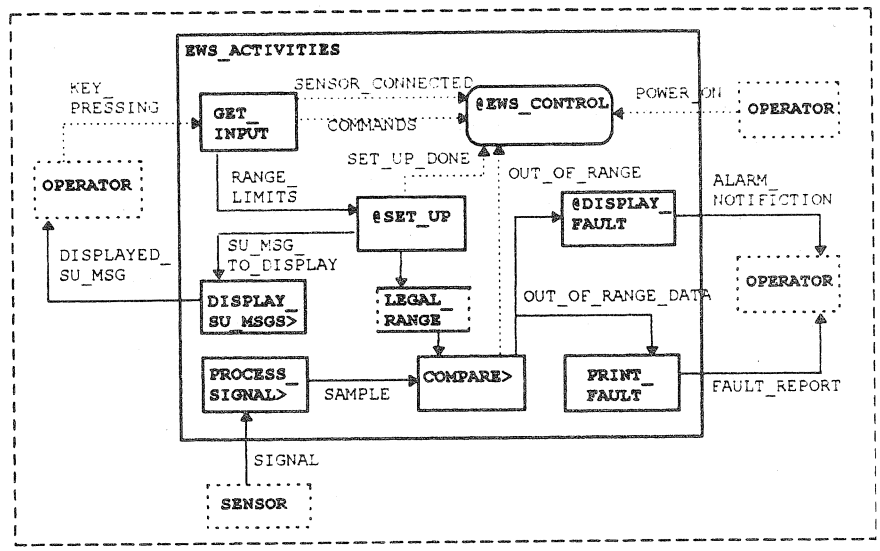

Figure B.3 Activity-chart EWS_ACTIVITIES.

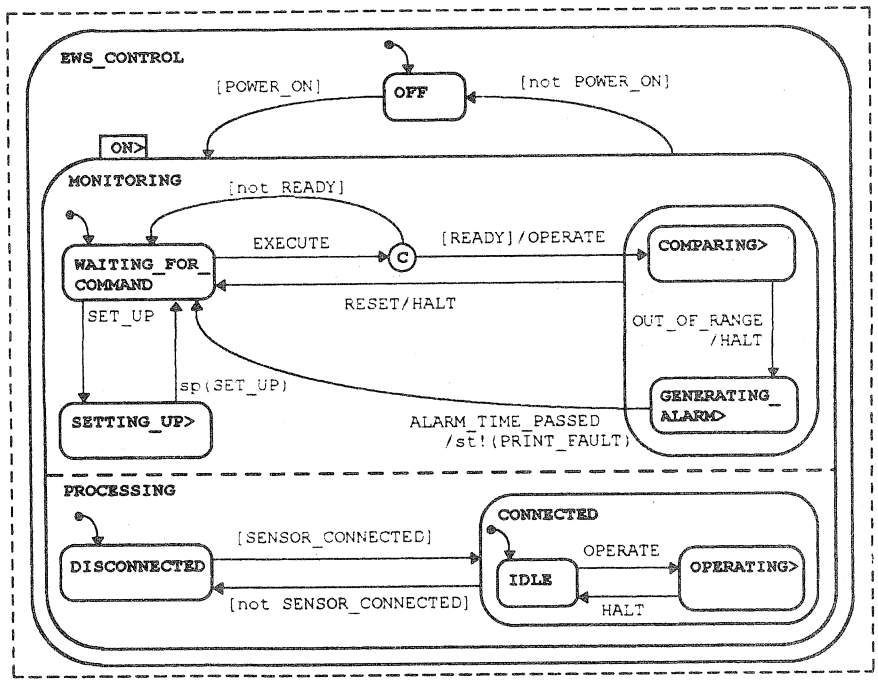

Figure B.4 Statechart EWS_CONTROL.

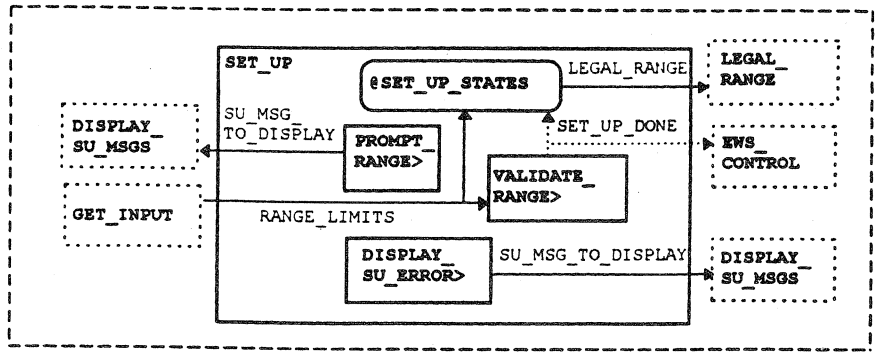

Figure B.5 Activity-chart SET_UP.

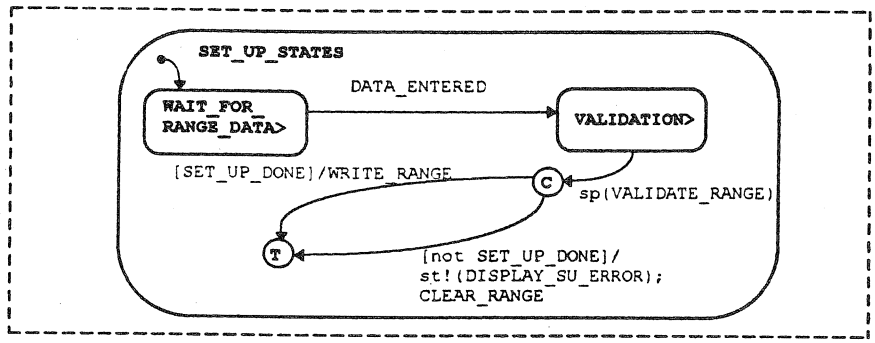

Figure B.6 Statechart SET_UP_STATES.

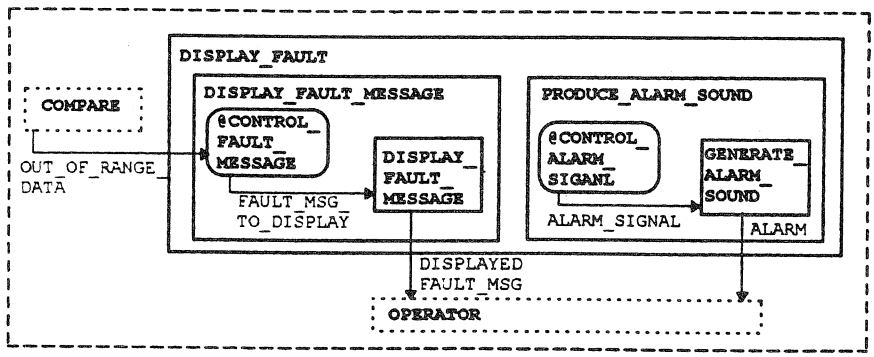

Figure B.7 Activity-chart DISPLAY_FAULT.

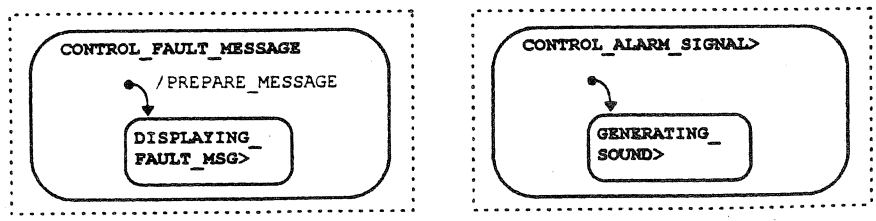

Figure B.8 Statecharts CONTROL_FAULT_MESSAGE and CONTROL_ALARM_SIGNAL.

B.2.3 The Data Dictionary

Modules

Module: EWS
Defined in Chart: EWS
Described by Activity-Chart: EWS_ACTIVITIES

Module: OPERATOR
Defined in Chart: EWS
Defined as: environment

Module: SENSOR
Defined in Chart: EWS
Defined as: environment

Activities and data-stores

Activity: COMPARE
Defined in Chart: EWS_ACTIVITIES
Termination Type: reactive controlled
Mini-spec:

```
wr(SAMPLE)/
     if ((SAMPLE < LEGAL_RANGE.LOW_LIMIT) or
          (SAMPLE > LEGAL_RANGE.HIGH_LIMIT)) then
               OUT_OF_RANGE;
               OUT_OF_RANGE_DATA.VALUE:=SAMPLE;
               OUT_OF_RANGE_DATA.LIMITS:=LEGAL_RANGE
     end if
```

Implemented by Module: CCU

Activity: CONTROL_ALARM_SIGNAL
Defined in Chart: DISPLAY_FAULT
Implemented by Module: CCU

Activity: CONTROL_FAULT_MESSAGE
Defined in Chart: DISPLAY_FAULT
Implemented by Module: CCU

Activity: DISPLAY_FAULT_MESSAGE
Defined in Chart: DISPLAY_FAULT
Implemented by Module: SCREEN

Activity: DISPLAY_SU_ERROR
Defined in Chart: SET_UP
Termination Type: procedure-like
Mini-spec:

```
SU_MSG_TO_DISPLAY:=`Range error; try again´
```

Activity: DISPLAY_SU_MSGS
Defined in Chart: EWS_ACTIVITIES
Termination Type: reactive controlled
Combinational assignments:

```
DISPLAYED_SU_MSG:=SU_MSG_TO_DISPLAY
```

Implemented by Module: SCREEN

Activity: GENERATE_ALARM_SOUND
Defined in Chart: DISPLAY_FAULT
Termination Type: reactive controlled
Implemented by Module: ALARM_SYSTEM

Activity: GET_INPUT
Defined in Chart: EWS_ACTIVITIES
Description: Transforms key pressing to data
Termination Type: reactive controlled
Implemented by Module: KEYBOARD

Data-store: LEGAL_RANGE
Defined in Chart: EWS_ACTIVITIES
Resides in Module: CCU

Activity: PRINT_FAULT
Defined in Chart: EWS_ACTIVITIES
Description: Issues fault data to the printer

Activity: PROMPT_RANGE
Defined in Chart: SET_UP
Termination Type: procedure-like
Mini-spec:

```
SU_MSG_TO_DISPLAY:=`Enter range limits´
```

Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES
Termination Type: reactive controlled
Mini-spec:

```
started/TICK;;
TICK/ $VALUE:=SIGNAL;
    SAMPLE:=COMPUTE($VALUE);-- ext. function
    sc!(TICK,SAMPLING_INTERVAL)
```

Implemented by Module: SIGNAL_PROCESSOR

Activity: SET_UP
Defined in Chart: EWS_ACTIVITIES
Termination Type: reactive self-terminated
Implemented by Module: CCU

Activity: VALIDATE_RANGE
Defined in Chart: SET_UP
Termination Type: procedure-like
Mini-spec:

```
fs!(SET_UP_DONE);
if RANGE_LIMITS.LOW_LIMIT<RANGE_LIMIT.HIGH_LIMIT
    then tr!(SET_UP_DONE)
end if
```

States

State: COMPARING
Defined in Chart: EWS_CONTROL
Activities in State: COMPARE (throughout)

State: CONTROL_ALARM_SIGNAL
Defined in Chart: CONTROL_ALARM_SIGNAL
Static Reactions:

```
ns/tr!(ALARM_SIGNAL);;
xs/fs!(ALARM_SIGNAL)
```

State: DISPLAYING_FAULT_MESSAGE
Defined in Chart: CONTROL_FAULT_MESSAGE
Activities in State: DISPLAY_FAULT_MESSAGE (throughout)

State: GENERATING_ALARM
Defined in Chart: EWS_CONTROL
Activities in State: DISPLAY_FAULT (throughout)

State: GENERATING_SOUND
Defined in Chart: CONTROL_ALARM_SIGNAL
Activities in State: GENERATE_ALARM_SOUND (throughout)

State: ON
Defined in Chart: EWS_CONTROL
Static Reactions: ns/fs!(SET_UP_DONE)
Activities in State: DISPLAY_SU_MSGS (throughout)

State: OPERATING
Defined in Chart: EWS_CONTROL
Activities in State: PROCESS_SIGNAL (throughout)

State: SETTING_UP
Defined in Chart: EWS_CONTROL
Static Reactions: ns/st!(SET_UP)

State: VALIDATION
Defined in Chart: SET_UP_STATES
Static Reactions: ns/st!(VALIDATE_RANGE)

State: WAIT_FOR_RANGE_DATA
Defined in Chart: SET_UP_STATES
Static Reactions: ns/st!(PROMPT_RANGE)

Events

Event: ALARM_TIME_PASSED
Defined in Chart: EWS
Definition: tm(en(GENERATING_ALARM),ALARM_DURATION)

Event: DATA_ENTERED
Defined in Chart: SET_UP_STATES
Definition: wr(RANGE_LIMITS)

Event: EXECUTE
Defined in Chart: EWS

Event: EXECUTE_KEY
Defined in Chart: EWS

Event: HALT
Defined in Chart: EWS_CONTROL

Event: OPERATE
Defined in Chart: EWS_CONTROL

Event: OUT_OF_RANGE
Defined in Chart: EWS_ACTIVITIES

Event: RESET
Defined in Chart: EWS

Event: RESET_KEY
Defined in Chart: EWS

Event: SET_UP
Defined in Chart: EWS

Event: SET_UP_KEY
Defined in Chart: EWS

Event: TICK
Defined in Chart: EWS_ACTIVITIES

Conditions

Condition: ALARM_SIGNAL
Defined in Chart: EWS

Condition: POWER_ON
Defined in Chart: EWS

Condition: READY
Defined in Chart: EWS_CONTROL
Definition: SET_UP_DONE and in(CONNECTED)

Condition: SET_UP_DONE
Defined in Chart: EWS_ACTIVITIES

Data-items

Data-Item: ALARM
Defined in Chart: EWS
Data-Type: real

Data-Item: ALARM_DURATION
Defined in Chart: EWS_CONTROL
Data-Type: real
Defined as: constant
Definition: 30.

Data-Item: DISPLAYED_FAULT_MSG
Defined in Chart: EWS
Data-Type: string

Data-Item: DISPLAYED_SU_MSG
Defined in Chart: EWS
Data-Type: string

Data-Item: FAULT_MSG_TO_DISPLAY
Defined in Chart: EWS
Data-Type: string

Data-Item: FAULT_REPORT_TO_PRINT
Defined in Chart: EWS
Data-Type: record
  Field Name: FAULT_TIME Field Type: TIME
  Field Name: FAULT_VALUE Field Type: integer
  Field Name: FAULT_RANGE Field Type: RANGE

Data-Item: FAULT_REPORT
Defined in Chart: EWS
Data-Type: string

Data-Item: HIGH_LIMIT_SLIDER
Defined in Chart: EWS
Data-Type: integer

Data-Item: LEGAL_RANGE
Defined in Chart: EWS_ACTIVITIES
Data-Type: RANGE

Data-Item: LOW_LIMIT_SLIDER
Defined in Chart: EWS
Data-Type: integer

Data-Item: RANGE_LIMITS
Defined in Chart: EWS
Data-Type: RANGE

Data-Item: OUT_OF_RANGE_DATA
Defined in Chart: EWS_ACTIVITIES
Data-Type: record
  Field Name: VALUE Field Type: integer
  Field Name: LIMITS Field Type: RANGE

Data-Item: SAMPLE
Defined in Chart: EWS
Data-Type: integer

Data-Item: SAMPLE_INTERVAL
Defined in Chart: EWS_ACTIVITIES
Data-Type: real
Defined as: constant
Definition: 2.

Data-Item: SIGNAL
Defined in Chart: EWS
Data-Type: bit-array 23 downto 0

Data-Item: SU_MSG_TO_DISPLAY
Defined in Chart: EWS
Data-Type: string

Actions

Action: CLEAR_RANGE
Defined in Chart: SET_UP_STATES
Definition:

```
LEGAL_RANGE.LOW_LIMIT:=0;
LEGAL_RANGE.HIGH_LIMIT:=0
```

Action: PREPARE_MESSAGE
Defined in Chart: CONTROL_FAULT_MESSAGE
Definition:

```
$VALUE_STR:=INT_TO_STRING(OUT_OF_RANGE_DATA.VALUE);
$OUT_STR:=STRING_CONCAT($VALUE_STR,' is out of range:\n');
$LOW_STR:=STRING_CONCAT(
    INT_TO_STRING(OUT_OF_RANGE_DATA.LIMITS.LOW_LIMIT),
      ` - `);
$HIGH_STR:=INT_TO_STRING(OUT_OF_RANGE_DATA.LIMITS.HIGH_LIMIT);
$RANGE_STR:=STRING_CONCAT($LOW_STR, $HIGH_STR);
FAULT_MSG_TO_DISPLAY:=STRING_CONCAT($OUT_STR,$RANGE_STR);
```

Action: WRITE_RANGE
Defined in Chart: SET_UP_STATES
Definition: LEGAL_RANGE:=RANGE_LIMITS

User-defined types

User-Defined Type: RANGE
Defined in Chart: EWS
Data-Type: record
  Field Name: LOW_LIMIT Field Type: integer
  Field Name: HIGH_LIMIT Field Type: integer

User-Defined Type: TIME
Defined in GDS: TIME_DEFS
Data-Type: record
  Field Name: HOURS Field Type: integer min=0 max=23
  Field Name: MINUTES Field Type: integer min=0 max=59
  Field Name: SECONDS Field Type: integer min=0 max=59

Information-flows

Information-Flow: ALARM_NOTIFICATION
Defined in Chart: EWS_ACTIVITIES
Consists of:

```
ALARM
DISPLAYED_FAULT_MSG
```

Information-Flow: COMMANDS
Defined in Chart: EWS
Consists of:

```
SET_UP
EXECUTE
RESET
```

Information-Flow: COMMAND_KEYS
Defined in Chart: EWS
Consists of:

```
SET_UP_KEY
EXECUTE_KEY
RESET_KEY
```

Information-Flow: DISPLAYED_MSGS
Defined in Chart: EWS
Consists of:

```
DISPLAYED_FAULT_MSG
DISPLAYED_SU_MSG
```

Information-Flow: KEY_PRESSING
Defined in Chart: EWS
Consists of:

```
COMMAND_KEYS
RANGE_SLIDERS
ENTER_KEY
SENSOR_CONNECTED_SWITCH
```

Information-Flow: MSGS_TO_DISPLAY
Defined in Chart: EWS
Consists of:

SU_MSG_TO_DISPLAY
FAULT_MSG_TO_DISPLAY

Information-Flow: MSGS_TO_PRINT
Defined in Chart: EWS
Consists of:

FAULT_REPORT_TO_PRINT

Information-Flow: RANGE_SLIDERS
Defined in Chart: EWS
Consists of:

LOW_LIMIT_SLIDER
HIGH_LIMIT_SLIDER

Information-Flow: USER_INPUT
Defined in Chart: EWS
Consists of:

COMMANDS
SENSOR_CONNECTED
RANGE_LIMITS

References

M. Alford, "SREM at the Age of Eight: The Distributed Computing Design System," *Computer* (April 1985), pp. 36–46.

B. Boehm, "Software Engineering," *IEEE Transactions on Computers* (December 1976), pp. 1226–1241.

———, "A Spiral Model of Software Development and Enhancement," *Computer* (May 1988), pp. 61–72.

G. Booch, *Object-Oriented Analysis and Design with Applications,* 2d ed., Benjamin/Cummings, Redwood City, CA, 1994.

W. Bruyn, R. Jensen, D. Keskar, and P. T. Ward, "ESML: An Extended Systems Modeling Language Based on Data Flow Diagram," *ACM Software Engineering Notes* **13** (January 1988), pp. 58–67.

J. Cameron, *JSP and JSD: The Jackson Approach to Software Development,* 2d ed., IEEE Computer Society Press, Los Alamitos, CA, 1989.

A. M. Davis, *Software Requirements: Analysis and Specification,* Prentice Hall, Englewood Cliffs, NJ, 1990.

T. DeMarco, *Structured Analysis and System Specification,* Yourdon Press, New York, 1978.

M. Dorfman and R. H. Thayer, *System and Software Requirements Engineering,* IEEE Computer Society Press, Los Alamitos, CA, 1990b.

———, *Standards, Guidelines, and Examples on System and Software Requirements Engineering,* IEEE Computer Society Press, Los Alamitos, CA, 1990a.

N. Francez, *Program Verification,* Addison-Wesley, Reading, MA, 1991.

H. Gomma, "Prototypes—Keep Them or Throw Them Away?," *State of the Art Report on Prototyping,* Pergamon Infotech Ltd., 1986.

———, *Software Design Methods for Concurrent and Real-Time Systems,* Addison-Wesley, Reading, MA, 1993.

——— and D. Scott, "Prototyping as a Tool in the Specification of User Requirements," *Proceedings of the Fifth International Conference on Software Engineering,* 1981, pp. 333–342.

D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* **8** (1987), pp. 231–274. (Preliminary version appeared as Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, Feb. 1984.)

———, *Algorithmics: The Spirit of Computing,* Addison-Wesley, Reading, MA, 1987; 2d ed., 1992a.

———, "On Visual Formalisms," *Communications of the ACM* **31** (1988), pp. 514–530.

———, "Biting the Silver Bullet: Toward a Brighter Future for System Development," *Computer* (January 1992b), pp. 8–20.

——— and E. Gery, "Executable Object Modeling with Statecharts," *Computer* (July 1997), pp. 31–42. (Also in *Proceedings of the 18th International Conference on Software Engineering,* Berlin, IEEE Press, March 1996, pp. 246–257.)

———, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering* **16** (1990), pp. 403–414.

——— and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology* **5** (October 1996), pp. 293–333. (Preliminary version appeared as Technical Report, I-Logix, Inc., 1989.)

——— and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems* (K. R. Apt, editor), NATO ASI Series, Vol F-13, Springer-Verlag, New York, 1985, pp. 477–498.

D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, New York, 1987.

International Telecommunication Union, *CCITT Specification and Description Language (SDL), ITU-T Recommendation Z.100,* 1995.

M. Jackson, *System Development,* Prentice-Hall, Englewood Cliffs, NJ, 1983.

I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Addison-Wesley, Wokingham, UK, 1992.

J. Z. Lavi and J. Kudish, "Systematic Derivation of Operational Requirements Using the ECSAM Method," *Proceedings of the IEEE Computer Society Israel 7th Conference on Computer-Based Systems and Software Engineering,* June 1996.

J. Z. Lavi and M. Winokur, "ECSAM—A Method for the Analysis of Complex Embedded Computer Systems and their Software," *Proceedings of the 5th Structured Techniques Association Conference,* Chicago, May 1989.

J. Z. Lavi, M. Winokur, R. Gallant, and J. Kudish, *Embedded Computer Systems Specification and Design—the ECSAM Approach,* IAI Technical Report, October 1992.

J. Loeckx and K. Seiber, *The Foundations of Program Verification,* John Wiley & Sons, New York, 1984.

D. McCracken and M. Jackson, "Life Cycle Concept Considered Harmful," *ACM Software Engineering Notes,* (April 1982), pp. 29–32.

Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification,* Springer-Verlag, New York, 1992.

Military Standard: Defense System Software Development, DOD-STD-2167A, U.S. Department of Defense, Washington, DC, February 1988.

Military Standard: Software Development and Documentation, MIL-STD-498, U.S. Department of Defense, Washington, DC, December 1994.

J. L. Peterson, *Petri Net Theory and Modeling of Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1981.

Rational Corp., *Documents on UML (the Unified Modeling Language),* Version 1.1 (http://www.rational.com/uml/), 1997.

W. W. Royce, "Managing the development of large software systems: Concepts and techniques," *Proceedings IEEE WESCON,* August 1970, pp. 1–9.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall, Englewood Cliffs, NJ, 1991.

B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling,* John Wiley & Sons, New York, 1994.

D. J. Smith, *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs using VHDL or Verilog,* Doone Publications 1996 (2d 1997, with minor revisions).

R. H. Thayer and M. Dorfman, *System and Software Requirements Engineering,* IEEE Computer Society Press, 1990.

P. T. Ward, "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software* (March 1989).

——— and S. J. Mellor, *Structured Development for Real-Time Systems,* Yourdon Press, New York, 1986.

D. P. Wood and W. G. Wood, "Comparative Evaluations of Four Specification Methods for Real-Time Systems," *Technical Report CMU/SEI-89-TR-36,* Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, 1989.

E. Yourdon and L. Constantine, *Structured Design,* Prentice-Hall, Englewood Cliffs, NJ, 1979.

P. Zave, "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM* (February 1984), pp. 104–118.

Index

ABOUT THE AUTHORS

David Harel is the Dean of Mathematics and Computer Science at the Weizmann Institute of Science in Rehovot, Israel. Dr. Harel is also the founder and chief scientist of I-Logix, Inc., the global firm that developed the STATEMATE system, and has been a visiting researcher and scientist at Carnegie-Mellon and Cornell Universities, as well as at Lucent Technologies, NASA, and IBM.

Michal Politi, formerly Vice President of Development for I-Logix Israel, Ltd. was responsible for the methodology and implementation of STATEMATE. She has headed computer research and development projects for the Israeli Defense Forces.