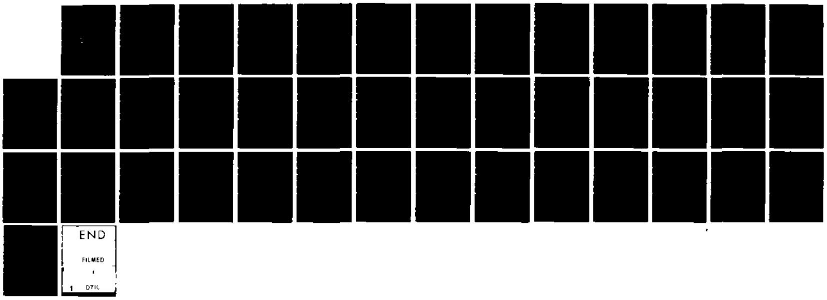
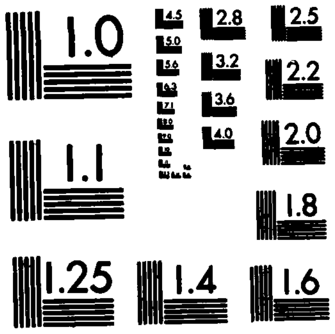
AD-A123 255   SUBTLE MANUAL(U) STANFORD UNIV CA DEPT OF COMPUTER SCIENCE M R GENESERETH ET AL. JAN 81 HPP-81-11-REV-3 N00014-81-K-0004   1/1

UNCLASSIFIED   F/G 9/2   NL


END
FILMED
1 DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(1)

Stanford Heuristic Programming Project
Memo HPP-81-11 (Working Paper)

January 1981
Revision 3.

SUBTLE Manual

Michael R. Genesereth
Milton Grinberg
Jay Lark

Department of Computer Science
Stanford University
Stanford, California 94305

Chapter 1 - Introduction

SUBTLE is a language for describing the design of digital circuits. The language enables one to specify complete or partial information about a device's structure (its parts and their interconnections), its behavior, and its teleology (arguments showing how the structure gives rise to the behavior).

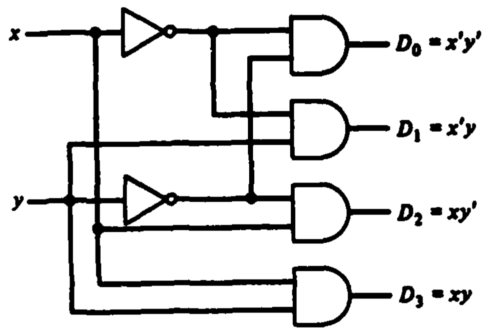
The syntax of SUBTLE is a prefix version of the language of predicate calculus and is identical to that used by MRS [Genesereth, Greiner, Smith]. The reader unfamiliar with MRS is encouraged to browse through the brief description in appendix 1 before starting on the manual. An interactive graphics interface coordinated with SUBTLE is currently under development [Lark] and should be available in the Spring of 1982.

There is also an interactive simulator/reasoner called SHAM. SHAM is essentially an "interpreter" for SUBTLE and can be used to explore the consequences of a design and thereby detect incompleteness or inconsistency. For further information on SHAM, the reader should see [Grinberg and Lark].

Chapter 2 of this manual describes SUBTLE's structural vocabulary, and chapters 3, 4, and 5 describe its behavioral vocabulary.. The behavior of a circuit can be characterized either by specifying its outputs directly in terms of its inputs or by describing the internal events that take place in normal operation. Chapter 3 introduces the vocabulary for time-dependent and time-independent I/O specification, and chapter 4 presents the vocabulary for describing events. Chapter 5 presents a programming language equivalent to the formalism of chapter 4. Appendix 1 is a brief introduction to the syntax of MRS; appendix 2 contains a complete SUBTLE dictionary, and appendix 3 presents SUBTLE descriptions of a number of common circuits.

Chapter 2 - Structure

The structure of a device is specified by describing its parts and their interconnections. The structure of each part can in turn be described until one reaches one's "primitive" components (which are usually characterized behaviorally). As an example of structural description in SUBTLE, consider the 2 by 4 decoder shown in figure 1.



(a) Logic diagram

x	y	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(b) Truth table

Figure 1 - A 2 by 4 decoder [Mano page 53]

2.1 Parts

The first step in describing a device is to enumerate the parts. In SUBTLE each part can be designated by either an atomic name (e.g. `A14`) or a functional description (e.g. `(amplifier-of M74)`). A good practice is to assign each part a unique name and equate the name of a part with its functional description where desired. For example, the following statement asserts that `A14` is the amplifer of `M74`.

```
(= (amplifier-of M74) A14)
```

{Footnote: There is a subtle distinction between these two possibilities: any statement including a named object (like A14) is a statement about the object itself; any statement including a functional description is a property of the role the object fills and must be true of any object that plays that role.}

The parts of a device are associated with the device itself via the subpart relation. For example, the components of the 2 by 4 decoder in figure 1 are related to the entire circuit (called `M74`) by the following statements.

```
(subpart inv-1 m74)
(subpart inv-2 m74)
(subpart and3-1 m74)
(subpart and3-2 m74)
(subpart and3-3 m74)
(subpart and3-4 m74)
```

For convenience, multiple part statements can be combined into a single statement using the `subpart*` relation, which associates a set of parts with the device they comprise. Thus, the above 6 statements could be rewritten as follows.

```
(subpart* inv-1 inv-2 and3-1 and3-2 and3-3 and3-4 M74)
```

The type of each part is declared using the type relation. The following assertions declare the types of the components: `inv-1` and `inv-2` are inverters; `and3-1`, `and3-2`, `and3-3`, and `and3-4` are 3-input and-gates.

```
(type inv-1 inv)
(type inv-2 inv)
(type and3-1 and3)
(type and3-2 and3)
(type and3-3 and3)
(type and3-4 and3)
```

Alternatively, one can take advantage of SUBTLE's part naming convention to declare types implicitly. If one uses a hyphenated name followed by a number (e.g. `and3-1`), this implies that the object so designated is of the type specified by the word before the hyphen (e.g. `and3-1` is of type `and3`). Given the names of the parts above, this convention eliminates the need for those statements altogether.

2.2 Connections

Every device in SUBTLE has zero or more inputs and outputs, and these "ports" are designated using the functions input and output. For example, `(input 2 and3-3)` would designate the second input of `and3-3`, and `(output 3 M74)` would designate the third input of `M74`. One can also assign mnemonic names to the inputs and outputs using equivalence statements like the following.

```
(= (enable M74) (input 1 M74))
```

The numbers of inputs and outputs are specified using the relations `sizein` and `sizeout`, e.g. the following statements declare `M74` to have 3 inputs and 4 outputs.

```
(sizein M74 3)
(sizeout M74 4)
```

Connections are made between the ports of devices. The next 18 assertions specify the wiring diagram for `M74`. For example, the first assertion states that the first input of `M74` is connected to the first input of `inv-1`.

```
(conn (input 1 m74) (input 1 inv-1))
(conn (input 1 m74) (input 2 and3-3))
(conn (input 1 m74) (input 1 and3-4))

(conn (input 2 m74) (input 2 and3-2))
(conn (input 2 m74) (input 2 and3-4))
(conn (input 2 m74) (input 1 inv-2))

(conn (input 3 m74) (input 3 and3-1))
(conn (input 3 m74) (input 3 and3-2))
(conn (input 3 m74) (input 3 and3-3))
(conn (input 3 m74) (input 3 and3-4))
```

```
(conn (output 1 inv-1) (input 1 and3-1))
(conn (output 1 inv-1) (input 1 and3-2))
(conn (output 1 inv-2) (input 2 and3-1))
(conn (output 1 inv-2) (input 1 and3-3))

(conn (output 1 and3-1) (output 1 m74))
(conn (output 1 and3-2) (output 2 m74))
(conn (output 1 and3-3) (output 3 m74))
(conn (output 1 and3-4) (output 4 m74))
```

For convenience, multiple connections to the same port can be collapsed into a single statement using the `conn*` relation. For example, the first three statements above can be summarized as follows.

```
(conn* (input 1 m74)
       (input 1 inv-1)
       (input 2 and3-3)
       (input 1 and3-4))
```

2.3 Generic Descriptions

The sections above discuss how a specific device can be described in terms of its specific parts. In electronics most circuit descriptions are generic: they describe a "prototypical" circuit of a given type, each instance of which has the stated structure.

In SUBTLE a generic circuit is treated as a set of physical devices, of which each instance is a member. The structure of a generic circuit is specified by first describing a typical member of the set and then stating that the circuit so described is a prototype for the entire set. For example, the description of `M74` can be made generic by adding the following statement.

```
(prototype M74 2x4decoder)
```

The import is that every circuit c declared to be a 2x4decoder (say with a statement of the form `(type c 2x4decoder)`) necessarily shares the same structure and properties.

However, this statement is not enough in itself. In the examples above, the prototype `M74` is described as having a part called `inv-1`. Nothing in the description distinguishes `inv-1` as a (possibly) unique part for each distinct 2x4decoder from `inv-1` as a single part that is shared by all 2x4decoders. In order to discriminate these cases, a generic description must also include a statement listing the parts not necessarily shared by each instance of the circuit. In the case of the 2x4decoder, the following statements would suffice.

```
(skolem M74 inv-1)
(skolem M74 inv-2)
(skolem M74 and3-1)
(skolem M74 and3-2)
(skolem M74 and3-3)
(skolem M74 and3-4)
```

A set of `skolem` statements can be abbreviated using the `skolem*` relation. In this way the above statements could be compressed to the single following statement.

```
(skolem* M74 inv-1 inv-2 and3-1 and3-2 and3-3 and3-4)
```

{Footnote: The entire description is equivalent to the following MRS statement.

```
(all M74 (exist inv-1 inv-2 and3-1 and3-2 and3-3 and3-4
               (if (mem M74 2x4decoder)
                   <description above>))))}
```

2.4 Parameterized Descriptions

As a result of regularity in the design of certain circuits, it is sometimes possible for the design to be scaled up in size. For example, the 2 by 4 decoder described above can be generalized to an arbitrary n by 2^n decoder. In SUBTLE circuits of variable size are encoded using "parameterized descriptions".

As before, one names a prototype as shown below.

```
(prototype m74 ixjdecoder)
```

For i by j decoder, there are two parameters, viz sizein (the number of inputs) and sizeout (the number of outputs). In principle only one is necessary, since one can be computed from the other; both are used here only for convenience. The general relationship between sizein and sizeout can be stated as follows.

```
(= (↑ 2 (sizein m74)) (sizeout m74))
```

Each i by j decoder has i inverters and j and gates. The parts cannot be individually named, because the number varies from device to device. However, they can be designated by inventing corresponding functions inverter and andgate that take an index and a device as arguments.

```
(if (<= 1 i (sizein m74)) (subpart (inverter i m74) m74))
(if (<= 1 j (sizeout m74)) (subpart (andgate j m74) m74))
```

The types of the parts can be declared using the access functions as follows.

```
(if (<= 1 i (sizein m74)) (type (inverter i m74) inv))
(if (<= 1 j (sizeout m74))) (type (andgate j m74) andn))
(if (<= 1 j (sizeout m74))) (size (andgate j m74) (size m74))
```

Finally, the connections can be stated.

```
(if (<= 0 i (size m74))
    (conn (input i m74) (input i (inverter i m74)))

(if (and (<= 1 m (↑ 2 (- (size m74) 1)))
         (<= 0 n (- (/ (size m74) i) 1)))
    (conn (output 1 (inverter i m74))
          (input i (andgate (+ (* n (↑ 2 i)) m) m74))))
```

```
(if (and (<= 1 m (↑ 2 (- (size m74) 1)))
         (<= 0 n (- (/ (size m74) i) 1)))
    (conn (input 1 m74)
          (input i (andgate (+ (* n (↑ 2 i)) m) m74))))

(if (<= 0 j (↑ 2 (size m74))
    (conn (output 1 (andgate j m74)) (output j m74))
```

An instance of a parameterized circuit is declared by stating its type and filling in the corresponding parameters. For example, a 3 by 8 decoder `M38` could be described as follows.

```
(type m38 ixjdecoder)
(sizein m38 3)
(sizeout m38 8)
```

Chapter 3 - I/O Specification

3.1 Simple I/O

The simplest form of behavioral specification is a set of rules relating a circuit's inputs to its outputs. As an example, consider the behavior of an inverter. When the input is on, the output is off; and vice-versa. This behavior is captured by the following two rules.

```
(if (on (input 1 $inv)) (off (output 1 $inv)))

(if (off (input 1 $inv)) (on (output 1 $inv)))
```

3.2 Time-dependant I/O

Note, however, that these rules say nothing about time, suggesting that the output changes instantaneously with the input. This is physically unrealistic; and while minute gate delays can often be ignored, timing considerations are sometimes crucial. The most basic temporal primitive in SUBTLE is the relation true. `(true p t)` is intended to mean that the proposition `p` is true at time `t`. Temporal behavior can be captured by relating the inputs of a circuit at a given time to its outputs after the appropriate interval. For example, the following statement means that, if the input to an inverter is on at one time instant, the output will be off after an interval equal to the gate delay of the inverter.

```
(if (true (on (input 1 $inv)) $t)
    (true (off (output 1 $inv)) (+ $t (delay $inv))))
```

In many digital circuits events are synchronized by clock pulses. In such designs time can be broken into discrete "cycles", and temporal descriptions can be written accordingly. For example, the following rendering of the above statement means that, if the input to an inverter is on at a given tick of the clock, the output will be off at the next tick.

```
(if (true (on (input 1 $inv)) $t)
    (true (off (output 1 $inv)) (+ $t 1)))
```

<Time intervals, periodic behavior, and frame axioms>

3.3 Procedural I/O Specification

A rule-based approach to I/O specification works well when there are few inputs and outputs and their relationship is straightforward. However, in many cases the relationship can be quite complex and may be more easily described in the form of a program.

In order to facilitate I/O specification in such cases, SUBTLE allows one to associate a Lisp subroutine with a device via a statement like the one below. The intended meaning of this statement is that, given a device's inputs, its output will be the same as that of the subroutine given the same inputs.

```
(function device-1 qr)
```

Since many devices have multiple outputs, it's necessary to augment Lisp to permit multiple return values. The values function illustrated in the definition below indicates that the arguments are to be returned as values of the subroutine.

```
(defun qr (x y)
       (values (quotient x y) (remainder x y)))
```

The setqs statement allows one to assign the values of a subroutine to a list of variables. After execution of the following statement, the variables `q` and `r` would have the values `1` and `11`, respectively.

```
(setqs (q r) (qr 23 12))
```

It is important to realize that Lisp I/O characterization need not correspond to the internal behavior of the device being described. Only the I/O must be accurate. Furthermore, no side effects (using `rplaca`, `rplacd`, or `setq`) are permitted outside the scope of the subroutine associated with a device. A more general approach to procedural description, allowing interaction and side effects, and intended to reflect the actual behavior of a set of interacting devices is presented in chapters 4 and 5.

Chapter 4 - Procedural Characterization of Behavior

This chapter introduces the SUBTLE vocabulary for describing the behavior of a set of interacting devices. The vocabulary is also commonly used to describe the internal behavior of a single device, since a single device is usually made up of a set of interacting parts.

In SUBTLE a program consists of a set of events, arbitrarily ordered by appropriate control and dataflow links. The structure of this "event graph" is described by MRS statements very similar to those used in describing the structure of a circuit. For visual immediacy, the structure can also be described in a two dimensional graphical representation. This chapter presents both the propositional and graphical languages, and chapter 5 presents a more traditional programming language equivalent.

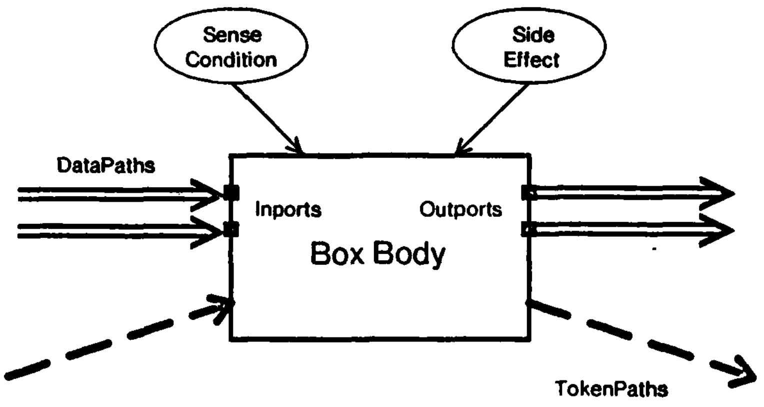
4.1 Action Boxes

An *action box* is the basic SUBTLE construct for describing some part of the internal or external behavior of a device. The execution of boxes is coordinated and controlled by the passing of tokens between boxes. Boxes can pass data to each other by data paths. Boxes can also interact through a global database. This chapter first describes the structure of boxes, and then goes on to other issues such as data paths, databases, and control.

Action boxes are referred to by name. A name can be any atomic symbol. A box is declared by the statement:

`(Box <agent> {<SUBTLE Box> or <set of SUBTLE Boxes>})` - <agent> has the given SUBTLE Boxes as part of its structure.

A box has several parts that can be classified according to function. The functional classes are I/O, preconditions, side-effects, and body. A box is represented graphically as:



4.1.1 Ports

Boxes can have an arbitrary number of input and output data *ports*. Each port is connected to a single data path (see section 4.2.1). This is the primary method for passing data into the body of the box. The semantics of an input data port are that data must be present on the data path for the box to start executing, else the box waits for the data to appear. Ports are represented in propositional form as:

`(Inport <port#> <box>)` – function that refers to the numbered input port of <box>.

`(Outport <port#> <box>)` – function that refers to the numbered output port of <box>.

4.1.2 Sense-conditions

Each box has zero or more *sense-conditions* that must be true in the current state of the world in order for the box to execute. Sense-conditions can be either *continuous* or *one-shot*. A one-shot sense-condition must be true at the moment the box attempts to start execution, while a continuous sense-condition must remain valid during the entire execution of the box. If a continuous sense-condition becomes untrue during execution the box stops executing. Sense-conditions are expressed as MRS propositions, and may contain variables. Sense-conditions are represented in propositional form as:

`(ContSenseCond <ActionBox> <expression>)` – continuous sense-condition of an action box.

`(OSSenseCond <ActionBox> <expression>)` – one-shot sense-condition of an action box.

4.1.3 Side-effects

Action boxes also have zero or more *side-effects* that become true in the current state of the world when a box has finished executing. Side-effects are expressed as MRS propositions in the following form.

```
(SideEffectAddition <ActionBox> <expression>)
```

4.1.4 Control

Each box also has connections for communicating control information. A box attempts to execute when it receives a control token (see section 4.4). When execution is complete, the box sends tokens down all control paths connected to it.

4.1.5 Body

The body of an action box can be a piece of Lisp code (a *Lisp box*) or a complete SUBTLE graph (a SUBTLE *subroutine*). The proposition necessary to specify the contents of an action box is:

`(Function <ActionBox> <FunctionDescriptor>)` – the functionality of <ActionBox> is given by <FunctionDescriptor>, where <FunctionDescriptor> is the name of either a Lisp or SUBTLE subroutine.

4.1.6 Lisp Boxes

The body of a Lisp box is a Lisp function. The number of arguments of the function must correspond to the number of data input ports on the box. On entering the procedure, the formal arguments are bound to the data from the appropriate DataPaths. The procedure returns its value(s) with the special SUBTLE control link *returns.* The statement to define the contents of a Lisp box is:

`(DefLambda <name> (arg`$_1$` ... arg`$_n$`) <body>)` – body of the Lisp box will be a lambda expression with the name <name>.

4.1.7 SUBTLE Subroutines

A SUBTLE *network* or *graph* can be formed by combining action boxes and connecting them with token paths and data paths. A graph is started into operation by a special set of tokens known as the *InitSet* of the graph. A graph can also have external data lines that are connected to the inputs and outputs of the agent or subroutine that contains it. The propositional representations for an InitSet are:

`(InitSet <agent> (set of <ActionBoxes>))` – specifies the boxes to receive tokens when the agent is started.

`(InitSet <subroutine> (set of <ActionBoxes>))` – specifies the boxes to receive tokens when the subroutine is passed a token.

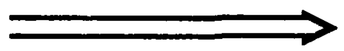
The content of a subroutine is a complete SUBTLE graph. When the subroutine is executed the graph is started as though it is a top level graph. The data inputs of the subroutine box are conceptually tied to the external input lines of the graph, in the same way an agent's inputs are tied to the external input lines of its graph.

4.2 Data Paths

The method of explicitly passing data from one SUBTLE box to another is through the use of a *DataPath.* A DataPaths connects an output port of a box with an input port of another box. It can also be used to connect different levels of a SUBTLE subroutine hierarchy. The plural form *DataPaths* is used to allow many sending ports to write data to each of the receiving ports. The propositional and graphical representations for DataPath and DataPaths are:

`(DataPath <sending port> <receiving port>)` – specifies that a data path exists between the sending port and the receiving port.

`(DataPaths {set of sending ports} {set of receiving ports})` – specifies that an or-in/and-out data path exists between each of the the sending and receiving ports.

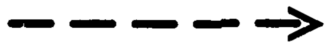


Each DataPaths is a unidirectional buffered connection between each of the sending ports and each of a set of receiving ports. The DataPaths is buffered to hold a single piece of data. Conceptually, this buffer is located just in front of each receiving port. When the data is read by the receiver its own buffer is emptied, but does not affect any other buffers on the same DataPaths. If any sending box attempts to send data down the DataPaths, the data will reach all receiving boxes connected to the path. If any old data already exists in the buffer it will be overwritten.

4.3 Control Links

An important concept in SUBTLE is that of *control*. A box can *execute* its activity when it has control, but can do nothing when it does not have control. Control is represented explicitly by *tokens*. When a box has a token it has control, and is said to be executing. Control passes from box to box by means of *TokenPaths*, which are unidirectional connections between boxes. By definition, a box retains control (a token) until it has finished executing. It then destroys any tokens it has and passes tokens to all boxes it is connected to by TokenPaths. A box need only have a single token to start executing, though it may have many at a time. A TokenPath is represented in propositional and graphical form as:

`(TokenPath <sender> <receiver>)` - specifies the control connections between boxes.



SUBTLE *control links* are used to coordinate and regulate the flow of tokens through the SUBTLE network, and to provide other control and data manipulation primitives. Control links can be thought of as special kinds of TokenPaths that have additional semantics. Individual links are referred to by name, which may be any atomic symbol, in the same way that action boxes are referred to.

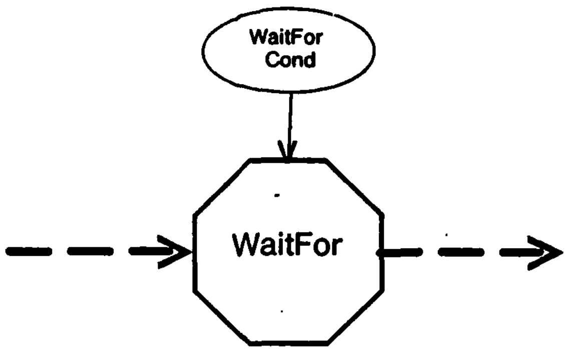
The rules for building a SUBTLE graph (a subroutine) are that it is to consist of action boxes that are linked to each other through control links. If normal control flow is desired the control link to use is a TokenPath. In this case a TokenPath is conceptually a token driver, that just outputs a token when it receives a token. More complex graphs can be built using other control links that alter the normal linear flow of control.

4.3.1 WaitFor

The *WaitFor* control link is used to halt control flow until a given sense-condition is true. If the sense-condition is true then control passes through the WaitFor, otherwise it goes into a wait state until for the sense-condition becomes true. The propositional and graphical representations for a WaitFor box are:

`(Link <WFLink> WaitFor)`

`(WFSensor <WFLink> <expression>)` – specifies the sense-condition that <WFLink> will wait for.



4.3.2 Condition

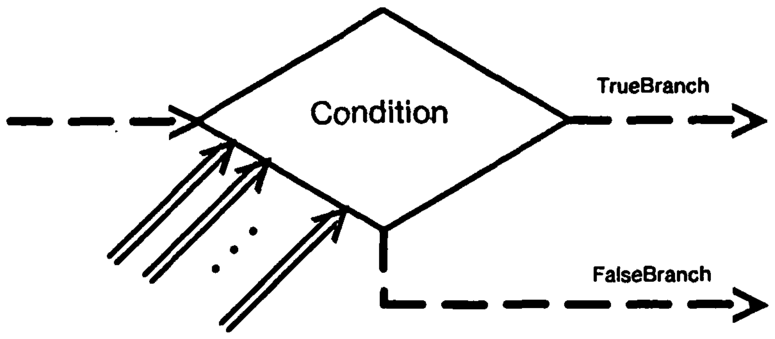
The *Condition* control link is used to make a branch in an otherwise linear control flow. When a Condition receives control it takes action based on its sense-condition. If the sense-condition is true it passes tokens along its *TrueBranch*, otherwise it passes tokens along its *FalseBranch*. TrueBranches and FalseBranches are special cases of TokenPaths. The propositional and graphical representations for a Condition link are:

`(Link <ConditionLink> Condition)`

`(ConditionSenseCond <ConditionLink> <expression>)` – declares pattern that <ConditionLink> will use to decide the branch to pass tokens along.

`(TrueBranch <ConditionLink> {set of <ActionBoxes>})` – <ConditionLink> will send tokens to indicated boxes if <ConditionLink>'s sense-condition is true.

`(FalseBranch <ConditionLink> {set of <ActionBoxes>})` – <ConditionLink> will send tokens to indicated boxes if <ConditionLink>'s sense-condition is not true.

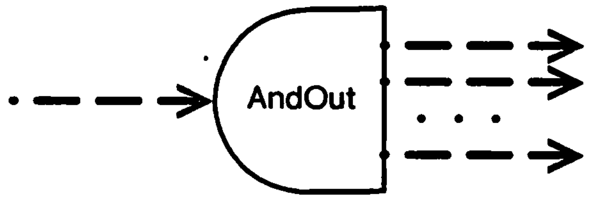


4.3.3 Branching Control Links

The branching control links are used to coordinate the flow of control between instruction streams that are to be executed in parallel. There are four different links that specify any or all coordination at a split or join in an execution stream.

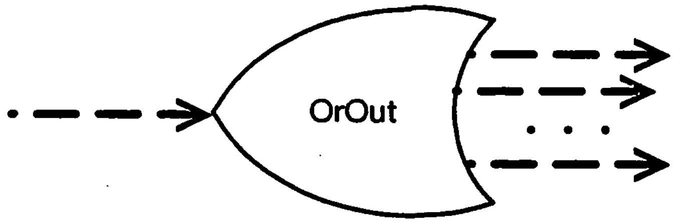
The *AndOut* control link is used to split the flow of control into two or more parallel streams. It is similar to the *Split* node of Procedural Nets [Sacerdoti]. When the AndOut is passed a token it passes tokens to all boxes connected to it by TokenPaths. (This is the default for all SUBTLE boxes.) The propositional and graphical representations for an AndOut link are:

```
(Link <AndOutLink> AndOut)
```



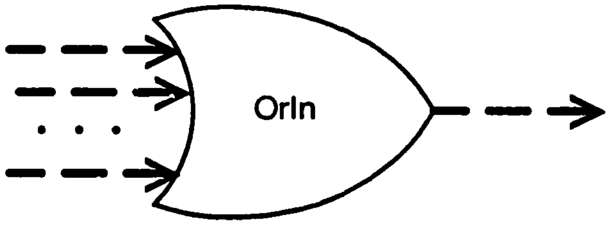
The *OrOut* control link is similar to an AndOut link except that when the OrOut receives a token it passes a token down only one of its connected TokenPaths, the choice being arbitrary. The propositional and graphical representations for an OrOut link are:

```
(Link <OrOutLink> OrOut)
```



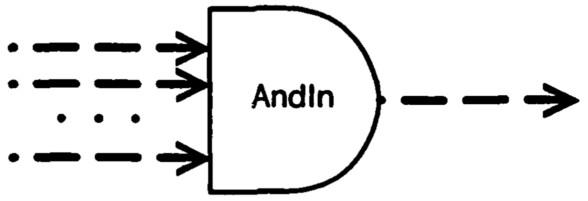
The *OrIn* control link is used to join parallel control streams. When an OrIn receives a token from any of its TokenPaths it passes tokens down its output TokenPaths. (This is the default for all SUBTLE boxes.) The propositional and graphical representations for an OrIn link are:

```
(Link <OrInLink> OrIn)
```



The *AndIn* control link is used to synchronize the execution of parallel control streams. It is similar in effect to the *Join* node of Procedural Nets. When the AndIn is passed a token it checks to see if it has received tokens from all boxes that can send it tokens. If it has received all tokens it passes tokens out, otherwise it goes into a wait state looking for the rest of the tokens. The propositional and graphical representations for an AndIn link are:

```
(Link <AndInLink> AndIn)
```



4.3.4 Interrupt and Resume

The *Interrupt* control link suspends or kills the execution of boxes when a given box gets control. The *Resume* control link allows an interrupted box to continue executing. The Interrupt and Resume links are different from the rest of the control links in that they don't have tokens passed directly to them. An Interrupt link is associated with a regular SUBTLE action box (the *From* box) that wishes to interrupt another SUBTLE action box (the *To* box). When the From box gets control the Interrupt link is activated and the To box is interrupted. A Resume link is similarly associated with a To and From box. When its From box passes control on, any interrupts on the To box are removed. In this way, a single From box can both Interrupt and Resume a To box.

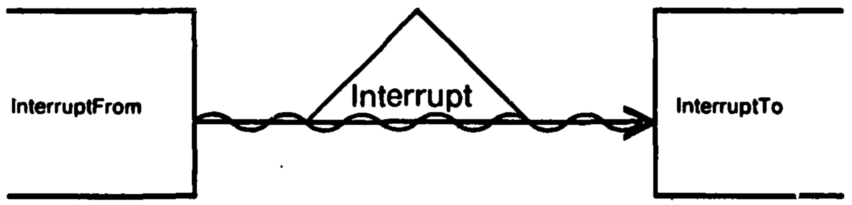
The type of interrupt that is given by an Interrupt link to its To box may be a *suspend* interrupt, which holds execution of the box until a Resume link removes the interrupt, or it may be a *kill* interrupt, which stops the box and destroys any tokens it has. A Resume link cannot restart a box that has been interrupted with a kill. The propositional and graphical representations for Interrupt and Resume links are:

`(Link <IntLink> Interrupt)`

`(IntFrom <IntLink> <sender>)` – <IntLink> asserts an interrupt when <sender> is executed.

`(IntTo <IntLink> <receiver>)` – <IntLink> asserts an interrupt to the indicated box.

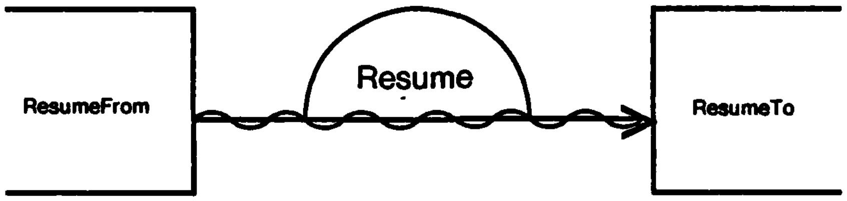
`(IntType <IntLink> {kill or suspend})` – <IntLink> assert an interrupt of the indicated type.



`(Link <ResumeLink> Resume)`

`(ResumeFrom <ResumeLink> <sender>)` – <ResumeLink> removes an interrupt when <sender> is finished executing.

`(ResumeTo <ResumeLink> <receiver>)` – <ResumeLink> removes an interrupt from the indicated box.



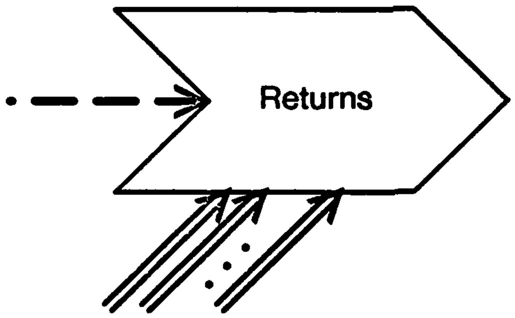
(These specifications are subject to revision in the near future.)

4.3.5 Returns

The *Returns* link is meant to be the SUBTLE analog of the Lisp return function. It takes its arguments and sends them to the output ports of the SUBTLE graph. It also stops processing of the current graph and detroys any tokens remaining in the graph. The propositional and graphical representations for a Returns link are:

`(Link <ReturnLink> Returns)`

`(ReturnList* <ReturnLink> <expr`$_1$`> ... <expr`$_n$`>)` – the named Return link returns the <expr$_i$> in order as the value of the current graph. The <expr$_i$> can be variables or inports of <ReturnLink>.

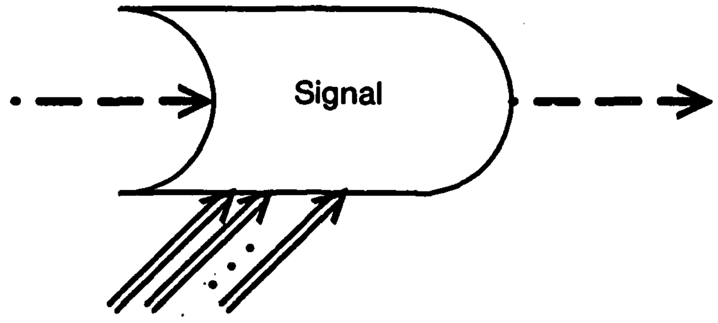


4.3.6 Signal

The *Signal* link is similar to the Returns link except that it does not stop the execution of the current graph. It is used to communicate partial results up the agent hierarchy, and for procedures that change their outputs in time. The propositional and graphical representations for a Signal link are:

`(Link <SignalLink> Signal)`

`(SignalList* <SignalLink> <expr`$_1$`> ... <expr`$_n$`>)` – the named Signal link sends the <expr$_i$> in order up graph hierarchy to the top agent.

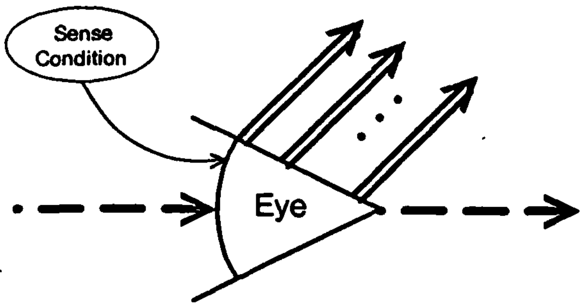


4.3.7 Eye

The *Eye* link is used to start a DataPath and initialize it with data from the current world. The arguments are expressions that correspond in number to the DataPaths coming into the link. When the link gets control data is placed on the appropriate DataPath according to the current state of the world, i.e. if ta expression is true then a "true" is put on the DataPath. If an expression contains variables the bindings for those variables that make the expression true are placed on the DataPath. The propositional and graphical representations for a Eye link are:

```
(Link <EyeLink> Eye)
```

`(EyeList* <EyeLink> <expr`$_1$`> ... <expr`$_n$`>)`



Chapter 5 - The Programming Language

While the vocabulary introduced in the last chapter is adequate for describing behavior, it can be somewhat tedious to use; and so SUBTLE includes a more traditional programming language equivalent. The language, Blisp, is a variant of Lisp, extended to include some additional control capabilities.

5.1 Introduction

Digital cicuit behavior is inherently parallel. Thus any language being used to express the behavior must provide some representation of parallelism. One of the important features of the SUBTLE graphic language is that this inherent parallelism is easily expressed. Representing this parallelism using a more traditional programming language is difficult. This is primarily due to traditional programming languages being one dimensional while parallelism requires two dimensions. SUBTLE works because it is expressed in two dimensions. The programming language being described in this section provides several functions that bring parallelism into this one dimensional form. Blisp not only augments the normal Lisp programming language with these special functions but also provides some constructs which can be utilized when programming in a parallel executing environment.

5.1.1 Control Flow

Most conventional programming languages have implicit control flow in the sequential ordering of the instructions. Under normal circumstances as one instruction completes, the next instruction in the sequential order is started. This ordering can be disrupted by special instructions like "GO". This flow of control from one instruction to the next can be metaphorically thought of as a token being passed from one instruction to the next. Only an instruction with a token is executing and there is at most one token in any program.

Blisp has this implicit control. However it also allows many tokens to be present in a program. Every instruction which has a token can be thought of as executing in parallel. The control an instruction exhibits in Blisp can be though of as either (1) passing a token to the next instruction, (2) destroying its token, or (3) splitting the token into many and then collapsing them back into a single token. The instruction path through which a token passes is referred to in the remainder of this chapter as the "execution path", "execution flow", or "control path".

5.1.2 The Equivalence of Blisp and SUBTLE Procedures

Since Blisp can have many instructions executing in parallel, it is a parallel programming language. However, because the coding m^thod for Blisp is that of a traditional programming language, some of the ease of expressiveness provided by the SUBTLE language is lost. Blisp is not intended to be used as a mapping from the SUBTLE structure into a programming language representation. It is intended to be an alternate form for describing behavior. It has been the experience of the designers that SUBTLE to Blisp conversion is a nontrivial task in all but the simplest examples because of the representation of the parallelism. However the reverse operation (i.e., Blisp to SUBTLE) is a much simpler task (excluding the layout difficulties). However, SUBTLE and Blisp are equivalent in power. All that can be represented in the SUBTLE can be represented in Blisp.

5.2 Blisp Functions

This section explains in detail the syntax, semantics, and use of each of the new procedures that augment Lisp to obtain Blisp. In many cases samples of how the compiler maps these instructions into the SUBTLE primitives is also provided.

5.2.1 WAITFOR

The WAITFOR statement provides a mechanism for suspending an execution path until an external condition has been met. An example of the use of this type of statement is a terminal which has a program that outputs characters to a main computer. It is normally waiting for another character to be typed in. When it recognizes the existence of the new character it run through code that transmits that character and then returns to the wait state.

The format of the Blisp WAITFOR statement is:

```
(WAITFOR <pred>)
```

The semantics of the WAITFOR is that it is waiting until the predicate becomes true at which time the WAITFOR completes allowing execution to continue at the next statement.

An example of the WAITFOR is:

```
(WAITFOR (on light))
(princ "light seen starting device")
```

which states that the WAITFOR statement maintains control until some other process makes the predicate (on light) true. Then the WAITFOR finishes and control is passed to the princ statement.

The compiler maps the WAITFOR Blisp statement into the following SUBTLE primitives:

```
(Link W1 WAITFOR)
(WFSensor W1 <pred>)
(TokenPath W1 E1)
```

where W1 is a gensymed name for the WAITFOR box and E1 is the gensymed name for the next sequential statement box (i.e., in the previous example the princ).

5.2.2 CONDITION

The Blisp CONDITION statement is a decision point from which one of two paths will be processed. It is the if-then construct for Blisp. The construct used to decide which path to take is an external condition of the same form as that in the WAITFOR. There are two possible formats for the CONDITION statement:

```
(CONDITION <pred> <then> <else>)
```

where if <pred> is true then the <then> function is processed otherwise the <else> function is processed. When the <then> or <else> parts completes, the CONDITION statement completes and execution continues at the next sequential statement.

```
(CONDITION <pred> <then>)
```

which is identical to the first form except that there is no <else> function. Thus if the <pred> is untrue then the CONDITION statement immediately completes.

There is a distinct difference between a CONDITION and a WAITFOR statement. A CONDITION statement immediately passes control on to one of two execution paths depending on the current truth of the predicate. A WAITFOR delays control until the predicate becomes true and then allows execution along the current execution path to continue.

An example of the CONDITION statement is:

```
(CONDITION (on light) (princ "light is on")
                      (progn (moveto lightswitch)
                             (princ "turning light on")
                             (push lightswitch)))
(princ "at next statement")
```

which has as its <then> clause '(princ "light is on")' and as its <else> clause '(progn (moveto lightswitch) . . . (push lightswitch))'. This senses if the light is on. If it is it prints that fact. Otherwise it moves to the lightswitch and turns it on. In either case it continues by printing "at next statement".

The compiler maps the Blisp CONDITION statement into the following SUBTLE primitives:

```
(Link C1 CONDITION)
(TrueBranch C1 T1)
(FalseBranch C1 E1)
(CondSensor C1 <pred>)
(Function T1 ...)*
(Function E1 ...)*
```

The * lines are compiled into their appropriate SUBTLE forms. C1, T1 and E1 are the gensymed names for the CONDITION statement box, the <then> statement box and the <else> statement box respectively.

5.2.3 CONDITIONS

The CONDITIONS statement is the Blisp version of the COND statement. It allows multiple predicate Blisp pairs. The statement that is associated with the first predicate that is true will be executed. The CONDITIONS statement returns the value of the executed statement. The format is:

```
(CONDITIONS (<pred1> <bstmt1>)
             . . .
            (<predn> <bstmtn>))
```

This is equivalent to the following CONDITION form:

```
(CONDITION <pred1> <bstmt1>
             . . .
                 (CONDITION <predn-1> <bstmtn-1> <bstmtn>))
```

For example:

```
(CONDITIONS ((low light) (adjust meter 3))
            ((medium light) (adjust meter 1))
            (T (princ "no adjustment necessary")))
```

adjusts the meter to 3 if the light is low, to 1 if the light is medium or prints a message that the meter doesn't need adjustment.

5.2.4 ALLALL

The ALLALL statement divides execution flow into many paths, all of which can be executed in parallel and all of which must finish before the ALLALL statement completes. This is one of the statements that provides parallelism for Blisp. The format of the ALLALL statement is:

```
(ALLALL <bstmt1> ... <bstmtn>)
```

where <bstmt$_i$>is a Blisp statement.

An example of the ALLALL statement is:

```
(ALLALL (setq a (- x y))
        (setq b (+ x y)))
(setq c (* a b))
```

The calculation of a and b can be accomplished in parallel but the execution of the (setq c ...) can not take place until both a and b have been calculated (i.e., the ALLALL statement completes).

The compiler maps the ALLALL statement into:

```
(Link A1 AndOut)
(Link A2 AndIn)
```

```
(Function bstmtname_1 ...)*
        .   .   .
(Function bstmtname_n ...)*
(TokenPath A1 bstmtname_1)
        .   .   .
(TokenPath A1 bstmtname_n)
(TokenPath bstmtname_1 A2)
        .   .   .
(TokenPath bstmtname_n A2)
```

where the * lines are compiled into their appropriate forms. Notice that the compiler maps this into two separate functions. One that divides control into the multiple paths and one that collapses these multiple paths back into one path.

5.2.5 ALLONE

The ALLONE statement is also used to divide execution into many paths. However, while ALLALL requires all the execution paths to finish, ALLONE only requires one of the execution paths to finish before it is complete. A use of the ALLONE statement is when there are several methods of obtaining an answer. All can be started, run in parallel, and the first that produces an answer allows the execution path to continue. (This description of the ALLONE statement permits the statement to be exited once. There is another instance of the ALLONE concept in which each completion of one of the paths causes execution to continue along the execution path. This form is not currently permitted in Blisp.) The format of the ALLONE is:

```
(ALLONE <bstmt_1> ...  <bstmt_n>)
```

where <bstmt$_i$> is a Blisp statement.

An example of the ALLONE statement is :

```
(setq b (ALLONE (sqrt1 a) (sqrt2 a) (sqrt3 a)))
```

There are three different square root routines that use different methods. The first one to complete provides a value to b and then the execution continues along the path.

The compiler maps the ALLONE statement into the following SUBTLE primitives:

```
(Link A1 AndOut)
(Link A2 OrIn)
(Function bstmtname_1 ...)*
        .   .   .
(Function bstmtname_n ...)*
(TokenPath A1 bstmtname_1)
        .   .   .
(TokenPath A1 bstmtname_n)
(TokenPath bstmtname_1 A2)
```

```
        . . .
(TokenPath bstmtname_n A2)
```

where the * lines are compiled into their appropriate forms. Notice that the compiler maps this into two separate functions. One that divides control into the multiple paths and one that collapses these multiple paths back into one path.

5.2.6 SETQS

The SETQS statement is a multiple setq statement that allows one to setq the values of one list into the variables of another. The format of the SETQS statement is:

```
(SETQS <varlist> <value1> ...  <valuen>)
```

where <varlist> is a list of variable name and each <value$_i$> is evaluated. The <value$_i$>'s are assigned to <varlist> in order. If the number of values is smaller than the <varlist> then the unused items in <varlist> are not assigned values. For example:

```
(SETQS (a b) 1 2)
```

is equivalent to:

```
(setq a 1)
(setq b 2)
```

and

```
(SETQS (a b c) 1 2)
```

is equivalent to:

```
(setq a 1)
(setq b 2)
```

5.2.7 RETURNS

The RETURNS statement is an exit out of a subroutine that returns a set of values which are the arguments to the calling routine. All called processes embedded within a execution path must exit using a RETURNS statement. If they don't then the execution path in which they were called is terminated. The format of the RETURNS is:

```
(RETURNS arg1 ...  argn)
```

For example:

```
(RETURNS 'a (list 'ab))
```

return to the calling program the set of values a and (ab).

5.2.8 DIE

The DIE statement is used to explicitly indicate that execution along an execution path terminates. Called procedures embedded in an execution path must return a value inorder for that execution path to continue. This statement provides explicitly the information that this will not happen. The format of the DIE statement is:

```
        (DIE)
```

For example:

```
    (COND (equal n 3) (RETURNS n 10)
                      (DIE))
```

will return the values (n 10) if n=3 otherwise it will terminate execution along the execution path of the calling routine.

5.2.9 INTERRUPT

The INTERRUPT statement allows one execution path to halt another execution path. This is only viable in an environment in which there is more than one execution path being processed in parallel. If the statement identified as being interrupted is not currently being executed then, in effect, this is a no-op (i.e., has no operational effect). The format of the Blisp INTERRUPT statement is:

```
    (INTERRUPT <label>)
```

where <label> is the label of the Blisp statement in the hierarchical scoping structure of the INTERRUPT statement. The Blisp statement associated with the label is halted by the INTERRUPT statement.

A RESUME statement (described next) allows the interruptted statement to continue executing. The lack of a RESUME statement, in effect, kills the execution path in which the interrupted statement is contained.

Consider these two partial programs:

```
                          . . .
program1       line (calculation x y)
                          . . .


                     . . .
program2       (interrupt line)
                     . . .
```

Suppose both programs 1 and 2 are running and program 1 is at the instruction at line. If program2 executes the (INTERRUPT line) instruction then the "calculation" statement will be halted. Hence that execution path in program 1 will be halted.

The compiler maps the Blisp INTERRUPT statement into the following SUBTLE primitives:

```
(Function I1 INTERRUPT)
(IntFrom I1 N1)
(IntTo I1 <label>)
```

where N1 is the name of the Function of the next Blisp statement following the INTERRUPT statement and <label> is the name of the Function associated with the interrupted statement.

5.2.10 RESUME

The RESUME statement is the counterpart of the INTERRUPT statement and is only meaningful if an INTERRUPT statement has been previously executed on the same label. The RESUME statement permits an interrupted statement to continue executing. If the statement is not in an interrupted state then it is in effect a no-op.

The format of the RESUME statement is:

```
(RESUME <label>)
```

where <label> is either a label or a Blisp statement within the scope of the RESUME statement.

If a <label> has both a RESUME and INTERRUPT statement associated with it, then it can be thought of as only being temporarily halted during the execution of some other part of the program. If their is no RESUME statement then it can be thought of as being terminated by the execution of another part of the program. Just having a RESUME statement is meaningless.

Consider the example from the INTERRUPT statement with the inclusion of a RESUME statement:

```
                          . . .
program1        line (calculation x y)
                          . . .


                        . . .
program2        (INTERRUPT line)
                 (princ "testing interrupt-resume")
                 (RESUME line)
                        . . .
```

The calculation at line is only temporarily halted while the message "testing interrupt-resume" is printed.

The compiler maps the Blisp RESUME statement into the following SUBTLE primitives:

```
(Function I1 RESUME)
(ResumeFrom I1 P1)
(ResumeTo I1 <label>)
```

where N1 is the name of the function of the Blisp statement preceding the RESUME statement and <label> is the name of the function associated with the interrupted statement.

5.2.11 WAIT

The WAIT statement stops execution along a path for a given number of time cycles. This is a way of internally delaying execution along a path. The format of the WAIT is:

```
(WAIT <n>)
```

where <n> is the number of time units execution is to be delayed.

For example:

```
(ringalarm)
(WAIT 10)
(condition (not (new-input)) (princ "are you still sleeping?"))
```

executes ringalarm (e.g., a routine that rings the bell at your terminal) and then waits for 10 time cycles before it checks to see if there is new input from the terminal. If there isn't, it writes the message to the terminal)

5.2.12 TIME

The TIME statement indicates the number of time cycles required before a Blisp statement can produce a value. The format of the statement is:

```
(TIME <n> <Blisp-stmt>)
```

where the <Blisp-stmt> will not complete until <n> cycles have passed from the time it was started. It is similar to the following:

```
(WAIT <n>)
(Blisp-stmt)
```

5.3 Monostable Multivibrator

The following set of Blisp programs describe the workings of a monstable multivibrator circuit. A monostable is used to generate a pulse of a given width by making one of its inputs high. The circuit behavior as follows: If either the clr line is low, input-a is high or input-b is low, then the circuit is in a reset condition with its primary output low and its inverted output high. If any of the inputs are in this state then the circuit can be said to be in initialized state. If any input changes from an initilized state into its inverted state while the other two inputs are in their non initialized states then a pulse is produced on the output line. After this is done one of the inputs must be reset to the initialized value. For example, if clr=high, a=low and b=low then the output=low (because b=low). If b then changes to high, a pulse will be produced on the output. No additional pulses can be triggered until one of the inputs is forced to its other state and then changed.

```
(defun mono1 (a b clr)
       (prog (q qbar)
             (condition (or a (not b) (not clr))
                        (progn (interrupt mono2)
                               (setq q low)
                               (setq qbar high)
                               (returns q qbar)
                               )
                        (die))
       ))

(defun mono2 (a b clr)
       (condition (or (and (falling a) clr b)
                      (and (not a) (rising clr) b)
                      (and (not a) clr (rising b)))
                  (returns 'high-pulse 'low-pulse)
                  (die))
        )

(defun mono (a b clr)
       (prog (pulse pulse-inv)
             (allany (setqs (pulse pulse-inv) (mono1 a b clr))
                     (setqs (pulse pulse-inv) (mono2 a b clr))
                     )
             (returns pulse pulse-inv)
        ))
```

Chapter 6 - Teleological Description

The teleology of a circuit is an argument explaining how the circuit's structure gives rise to its behavior. The word *teleology* is used because such arguments implicitly include the purpose of each of the circuit's components.

6.1 Justifications

The *justification* of an expectation about a circuit's behavior is a trace of the reasoning steps necessary to prove the expectation given the circuit's structure and its inputs. In SUBTLE each step in a justification is encoded as a separate statement of the form `(just q m p1 . . . pn)`, where `q` is the conclusion, `m` is the reasoning method, and `p1`, . . ., `pn` are the premises. For example, the statement below presents an argument explaining why the output of `M74` is expected to be on using the backward chaining method `bc-truep`.

```
(just (on (output 1 M74)) bc-truep
              (if (and (conn $x $y) (on $x)) (on $y))
              (and (conn (output 1 and3-1) (output 1 M74))
                   (on (output 1 and3-1))))
```

Currently, SUBTLE includes the following set of reasoning methods.

`ex-truep` - universal instantiation and existential generalization
`bc-truep` - backward chaining
`truep-and` - conjunction
`truep-or` - disjunction
`assume` - assumption

This set was chosen because it corresponds to the basic inference methods in SHAM.

6.2 Design Knowledge and Meta-Teleology

<to be written>

Chapter 7 - Conclusion

<to be written>

References

MRS paper

Petri Nets

Mano

CSA

Procedural Nets

SDL and ADLIB

Appendix 1 - The Syntax of MRS

MRS is a prefix version of the language of predicate calculus.

A1.1 Symbols

There are two types of symbols in MRS, viz. *variables* and *constants*. Variables are useful for stating facts about all members of a set or for declaring the existence of an object without naming it. The use of variables is elaborated below in the discussion of quantified propositions.

There are three different types of constant symbols. *Object symbols* name specific objects or concepts in the world being described, e.g.

```
M74
inverterB
2x4decoders (the set thereof)
AND3-3
Stanford
Kennedy
```

Function symbols are intended to represent functions on the objects of the world, e.g.

```
sizein
type
president-of
height-of
```

Relation symbols represent relations between objects of the world, e.g.

```
subpart
>
older-than
neighbor
```

A1.2 Terms

In MRS one can also designate objects by combining these symbols into more complex expressions, called *terms*. All variables and constants are terms by definition. In addition, given an n-ary function symbol `f` and n terms `t1`, . . ., `tn`, then the expression `(f t1 . . . tn)` is also a term. For example, the following expressions are legal terms.

```
(subpart M74)
(sizein AND3-3)
(president-of Stanford)
(+ 2 2)
(height-of (president-of Stanford))
(* (+ 2 2) 3)
```

A1.3 Atomic Propositions

Facts can be stated withn MRS in the form of *propositions*. Given an n-ary relation symbol **r** and n terms **t1, . . . , tn**, the expression **(r t1 . . . tn)** is an *atomic proposition*. The following are examples.

```
(subpart M74 AND3-3)
(neighbor Palo-Alto Menlo-Park)
(> (* 2 3) (+ 2 3)).
```

A1.4 Logical Expressions

Unfortunately, not all facts are so simple. One often needs to express negations (e.g. "Lyman is not the president of Stanford), disjunctions (e.g. "Either Lyman is president or Kennedy is president"), and contingencies (e.g. "If George is at home, he must be sick"). In MRS facts like these can be written by relating the appropriate atomic propositions via *logical symbols* such as **not**, **and**, **or**, and **if**. For example, these sentences could be written as follows.

```
(not (= (sizein AND3-3) 2))
(not (= (president-of Stanford) Lyman))
(or (= (president-of Stanford) Lyman)
    (= (president-of Stanford) Kennedy))
(if (location george home) (sick george))
```

A1.5 Quantified Propositions

Finally, there are quantified propositions. With the syntax given so far, one can only write facts by naming the objects involved. There's no simple way to talk about all the members of a set or state the existence of an object without naming it. Quantifiers enable one to state facts like "All apples are red" and "There's a doctor in the house". There are two quantifiers in MRS, viz. **all** and **exist**. The proposition **(all x1 . . . xn (p x1 . . . xn))** states that **(p x1 . . . xn)** is true for all possible values of the variable symbols **x1, . . . , xn**. The proposition **(exist x1 . . . xn (p x1 . . . xn))** states that there exist objects **a1, . . . , an** for which **(p a1 . . . an)** is true. For example, the first proposition below states that all apples are red, and the second says that there's a doctor in the house. Quantified propositions can also occur within non-atomic propositions, as in the last two examples.

```
(all x (if (mem x apples) (color-of x red)))
(exist x (and (mem x doctors) (location-of x house)))
(or (all x (apple x)) (some x (pear x)))
(all x (exist y (> y x)))
```

Multiple variables of the same type can be declared within a single quantified proposition, as illustrated below. Note, however, that the ordering of quantifiers is essential whenever a quantified proposition is nested within another. For example, the last two propositions mean two very different things. Information about the order of nesting of quantified propositions is sometimes referred to as *skolem information*.

```
(all h r (if (and (horse h) (rabbit r)) (can-outrun h r)))
(exist x y (and (= (+ x 1) y) (= (* 2 x) y)))
(all x (exist y (loves x y)))
(exist y (all x (loves x y)))
```

Two useful syntactic features are illustrated in the examples below. The first is the use of the prefix characters $ and ? to denote universal variables and leftmost existential variables. Each member of the following pairs of assertions is equivalent to the other.

```
(all x (if (neighbor x Bertram) (neighbor x Beatrice)))
(if (neighbor $x) (neighbor $x Beatrice))
(exist x (and (apple x) (color-of x red)))
(and (apple ?x) (color-of x red))
```

Second, a function in MRS can also be used as a relation in propositions where the value is specified as the last argument.

```
(= (sizein AND3-3) 3)
(sizein AND3-3 3)
```

Appendix 2 - SUBTLE Dictionary

A2.1 Structural Vocabulary:

`conn` - `(conn <x> <y>)`
States that the "port" (input or output line) `<x>` is connected to to the port `<y>`. Note that the connection is conceptual; at one level of detail two ports may be simply connected whereas at a lower level of detail the connection is described more completely as consisting of some wires or solders together with their connections.

`conn*` - `(conn* <x1> <y1> ... <yn>)`
Is equivalent to `(conn <x1> <y1>), ... , (conn <x1> <yn>)`.

`input` - `(input <i> <device>)`
Refers to the `<i>`th input line of object `<device>`.

`output` - `(output <i> <device>)`
Refers to the `<i>`th output line of object `<device>`.

`subpart` - `(subpart <part> <device>)`
States that the object `<part>` is a component of `<device>`.

`subpart*` - `(subpart* <part1> ... <partn> <device>)`
Is equivalent to `(part <part1> <device>) ... (part <partn> <device>)`

`prototype` - `(prototype <name> <type>)`
States that `<name>` is a prototype for the generic circuit of type `<type>`. All of the properties of `<name>` are "inherited" by every instance of the type.

`skolem` - `(skolem (<x1> ... <xm>) (<y1> ... <yn>))`
States that `<y1> ... <yn>` are existential variables governed by the universal variables `<x1> ... <xm>`. The statement is useful in specifying the parts of a description that are unique to each instance of a generic circuit.

`type` - `(type <device> <type>)`
States that `<device>` is an object of type `<type>`.

A2.2 Behavioral Vocabulary:

`Box - (Box <agent> {<ActionBox> or <set of ActionBoxes>})`
`<agent>` has the given SUBTLE Boxes as part of its structure.

`ConditionSenseCond - (ConditionSenseCond <ConditionLink> <expression>)`
Declares pattern that `<ConditionLink>` will use to decide the branch to pass tokens along.

`ContSenseCond - (ContSenseCond <ActionBox> <expression>)`
Continuous sense-condition of an action box.

`DataPath - (DataPath <sending port> <receiving port>)`
Specifies that a data path exists between the sending port and the receiving port.

`DataPaths - (DataPaths {set of sending ports} {set of receiving ports})`
Specifies that an or-in/and-out data path exists between the sending and receiving ports.

`DefLambda - (DefLambda <name>` $(arg_1 \ldots arg_n)$ `<body>)`
Specifies that the Function of a box will be a lambda with name `<name>`.

`FalseBranch - (FalseBranch <ConditionLink> {set of <ActionBoxes>})`
`<ConditionLink>` will send tokens to indicated boxes if `<ConditionLink>`'s database sense-condition is not true.

`Function - (Function <ActionBox> <FunctionDescriptor>)`
The functionality of `<ActionBox>` is given by `<FunctionDescriptor>`, where `<FunctionDescriptor>` is the name of either a Lisp or SUBTLE subroutine.

`InitSet - (InitSet <agent> {set of <ActionBoxes>})`
Specifies the boxes to receive tokens when the agent is started for the first time.

`InitSet - (InitSet <subroutine> {set of <ActionBoxes>})`
Specifies the boxes to receive tokens when the subroutine is passed a token.

`Inport - (Inport <port#> <ActionBox>)`
Function that refers to the numbered input port of `<ActionBox>`.

`IntFrom - (IntFrom <IntLink> <sender>)`
`<IntLink>` asserts an interrupt when `<sender>` is executed.

`IntTo - (IntTo <IntLink> <receiver>)`
`<IntLink>` asserts an interrupt to the indicated action box.

`IntType - (IntType <IntLink> {kill or suspend})`
`<IntLink>` assert an interrupt of the type indicated.

`Link - (Link <ControlLink> <type>)`
`<ControlLink>` is a control link of type `<type>`.

`OSSenseCond - (OSSenseCond <ActionBox> <expression>)`
One-shot sense-condition of an action box.

`Outport - (Outport <port#> <box>)`
Function that refers to the numbered output port of `<box>`.

`ResumeFrom - (ResumeFrom <ResumeLink> <sender>)`
`<ResumeLink>` removes an interrupt when `<sender>` is finished executing.

`ResumeTo - (ResumeTo <ResumeLink> <receiver>)`
`<ResumeLink>` removes an interrupt from the indicated action box.

ReturnList* - (ReturnList* <ReturnLink> <expr$_1$> ... <expr$_n$>)
The named Return link returns the <expr$_i$> in order as the value of the current graph. The <expr$_i$> can be variables or inports of `<ReturnLink>`.

`RunTime - (RunTime <ActionBox> <time>)`
Time that `<ActionBox>` requires to execute.

SignalList* - (SignalList* <SignalLink> <expr$_1$> ... <expr$_n$>)
The named Signal link send the <expr$_i$> in order up graph hierarchy to the top agent.

`TokenPath - (TokenPath <sender> {<receiver> or <set of receivers>})`
Specifies the control connections between SUBTLE boxes.

`TrueBranch - (TrueBranch <ConditionBox> {set of <ActionBoxes>})`
`<ConditionBox>` will send tokens to indicated boxes if `<ConditionBox>`'s sense-condition is true.

TruepList* - (TruepList* <TruepLink> <expr$_1$> ... <expr$_n$>)
Fetches the value of the <expr$_i$> from the current local context and places the results on the Truep link's outports.

`WFSenseCond - (WFSenseCond <WFLink> <expression>)`
Declares pattern that `<WFLink>` will wait to occur.

A2.3 Blisp: Lisp + the following "functions"

`ALLALL - (ALLALL ARG1 ... ARGn)`

Each `ARGi` is started and, when all have finished, control is passed to the statement following the `ALLALL`. If any fail to complete (i.e., it dies), the `ALLALL` statement fails and control terminates along that path.

`ALLONE - (ALLONE ARG1 ... ARGn)`

Each `ARGi` is started and when any have finished, control is passed to the statement following the `ALLONE`. If all fail to complete (i.e., they all die), the `ALLONE` statement fails and control terminates along that path.

`CONDITION -`

```
(1) (CONDITION ARG1 ARG2)
(2) (CONDITION ARG1 ARG2 ARG3)
```

1. If `ARG1` is true then execute `ARG2`.
2. If `ARG1` is true then execute `ARG2` otherwise execute `ARG3`.

In both cases continue on.

`CONDITIONS - (CONDITIONS (pred statement) ... (pred statement))`

This is the Blisp version of the `COND` statement. The first predicate which is true causes its paired statement to gain control and execute. If the statement dies then control flow ends on this path.

`DIE - (DIE)`

Subroutine stops unnaturally without returning a value. Control flow is stopped along this path.

`INTERRUPT - (INTERRUPT ARG1)`

The program named by `ARG1` is forced to die without returning a value.

`RETURNS - (RETURNS item-1 ... item-n)`

Return a list of values from a subroutine.

`SETQS - (SETQS (list1) (list2))`

Each of the individual items in list1 (unevaluated) is assigned a value from list2 (evaluated).

`TIME - (TIME <time> <Blisp-statement-1> ... <Blisp-statement-n>)`

The simulator will not produce results from the Blisp-statements for `<time>` cycles. The `<Blisp-statement-i>` can be any Blisp statement.

`WAIT - (WAIT <n>)`

Delay for `<n>` time cycles before continuing with execution of this control path.

`WAITFOR - (WAITFOR ARG1)`

When `ARG1` becomes true, execution can continue otherwise it waits for `ARG1` to become true. NO statement after the `WAITFOR` will be executed until the `WAITFOR` condition becomes true.

Appendix 3 - Library of Standard Circuits

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;               the behavioral description of an inverter          ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $inv inv) (on (input 1 $inv)))
    (off (output 1 $inv)))

(if (and (type $inv inv) (off (input 1 $inv)))
    (on (output 1 $inv)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;               the behavioral description of an and-gate          ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $and and-gate) (on (input 1 $and)) (on (input 2 $and)))
    (on (output 1 $and)))

(if (and (type $and and-gate) (off (input 1 $and)))
    (off (output 1 $and)))

(if (and (type $and and-gate) (off (input 2 $and)))
    (off (output 1 $and)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;          the behavioral description of a 3-input and-gate        ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $and3 and3)
    (on (input 1 $and3)) (on (input 2 $and3)) (on (input 3 $and3)))
    (on (output 1 $and3)))

(if (and (type $and3 and3) (off (input 1 $and3)))
    (off (output 1 $and3)))

(if (and (type $and3 and3) (off (input 2 $and3)))
    (off (output 1 $and3)))

(if (and (type $and3 and3) (off (input 3 $and3)))
    (off (output 1 $and3)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;              the behavioral description of an or-gate          ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $or or-gate) (or (on (input 1 $or)) (on (input 2 $or))))
    (on (output 1 $or)))

(if (and (type $or or-gate) (off (input 1 $or)) (off (input 2 $or)))
    (off (output 1 $or)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;              the behavioral description of an xor-gate         ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(if (and (type $xor xor-gate) (off (input 1 $xor)) (off (input 2 $xor)))
    (off (output 1 $xor)))

(if (and (type $xor xor-gate) (off (input 1 $xor)) (on (input 2 $xor)))
    (on (output 1 $xor)))

(if (and (type $xor xor-gate) (on (input 1 $xor)) (off (input 2 $xor)))
    (on (output 1 $xor)))

(if (and (type $xor xor-gate) (on (input 1 $xor)) (on (input 2 $xor)))
    (off (output 1 $xor)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    Structural Description of a full adder. See Mano page 20.        ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(prototype fa-1 full-adder)
(sizein fa-1 3)
(sizeout fa-1 2)

(subpart* xor-1 and-1 xor-2 and-2 or-1 fa-1)
(type xor-1 xor-gate)
(type and-1 and-gate)
(type xor-2 xor-gate)
(type and-2 and-gate)
(type or-1 or-gate)

(conn* (input 1 fa-1) (input 1 xor-1) (input 1 and-1))
(conn* (input 2 fa-1) (input 2 xor-1) (input 2 and-1))
(conn* (input 3 fa-1) (input 1 xor-2) (input 1 and-2))

(conn* (output 1 xor-1) (input 2 xor-2) (input 2 and-2))
(conn* (output 1 and-1) (input 1 or-1))
(conn* (output 1 and-2) (input 2 or-1))

(conn* (output 1 xor-2) (output 1 fa-1))
(conn* (output 1 or-1) (output 2 fa-1))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;    2x4 is an implementation of a 2x4-decoder.  See Mano page 53. ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(prototype 2x4 2x4-decoder)
(sizein 2x4 3)
(sizeout 2x4 4)

(subpart* inv-1 inv-2 and3-1 and3-2 and3-3 and3-4 2x4)
(type inv-1 inv)
(type inv-2 inv)
(type and3-1 and3)
(type and3-2 and3)
(type and3-3 and3)
(type and3-4 and3)

(conn* (input 1 2x4) (input 1 inv-1) (input 2 and3-3) (input 1 and3-4))
(conn* (input 2 2x4) (input 2 and3-2) (input 2 and3-4) (input 1 inv-2))
(conn* (input 3 2x4) (input 3 and3-1) (input 3 and3-2)
                     (input 3 and3-3) (input 3 and3-4))
(conn* (output 1 inv-1) (input 1 and3-1) (input 1 and3-2))
(conn* (output 1 inv-2) (input 2 and3-1) (input 1 and3-3))
(conn* (output 1 and3-1) (output 1 2x4))
(conn* (output 1 and3-2) (output 2 2x4))
(conn* (output 1 and3-3) (output 3 2x4))
(conn* (output 1 and3-4) (output 4 2x4))
```