MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A. I. Memo 1227 **April, 1990**

The Behavior Language; User's Guide*

Rodney A. Brooks

Abstract

The Behavior Language is a rule-based real-time parallel robot programming language originally based on ideas from [**Brooks 86**], [**Connell 89**], and [**Maes 89**]. It compiles into a modified and extended version of the subsumption architecture [**Brooks 86**] and thus has backends for a number of processors including the Motorola 68000 and 68HC11, the Hitachi 6301, and Common Lisp. Behaviors are groups of rules which are activatable by a number of different schemes. There are no shared data structures across behaviors, but instead all communication is by explicit message passing. All rules are assumed to run in parallel and asynchronously. It includes the earlier notions of inhibition and suppression, along with a number of mechanisms for spreading of activation.

*Support for this research was provided in part by the University Research Initative under Office of Naval Research contract N00014–86–K–0685, in part by the Advanced Research Projects Agency under Office of Naval Research contract N00014–85–K–0124 and in part by a gift from Siemens.

1 Introduction

The subsumption architecture was described initially in [**Brooks 86**] and later modified in [**Brooks 89**] and [**Connell 89**]. The subsumption compiler compiles augmented finite state machine (AFSM) descriptions into a special purpose scheduler to simulate parallelism and a set of finite state machine simulation routines. This is a dynamically retargettable compiler that has backends already for a number of processors, including the 68000, the 68HC11, and the 6301. The subsumption compiler takes a source file as input, and depending on the target machine either produces an assembly source file or an incore data structure that can be assembled by some other assembler.

The behavior language was inspired by [**Maes 89**] as a way of grouping AFSMs into more manageable units with the capability for whole units being selectively activated or de-activated. In fact, AFSMs are not speficied directly, but rather rule sets of real-time rules compile into AFSMs in a one-to-one manner. Sharing of registers and monostables within the AFSMs produced by a single rule set, or behavior, is the norm. The behavior compiler is machine independent and compiles into an intermediate file of subsumption AFSM specifications. The subsumption compiler can then be used to compile to the various targets. Some enhancements were made to the original subsumption language in order to support the behavior language.

The behavior language is sometimes referred to as the *new subsumption*. A behavior language program appears as groupings of real-time rules, which are written in a subset of Lisp, and which run in parallel.

All code for the behavior and subsumption compilers was written in Common Lisp.

2 The User Interface

There are three forms for interacting with the behavior compiler at Lisp top level. These specify the target machine, invoke the behavior compiler, and invoke the subsumption compiler respectively.

2.1 Specification

*** `(set-current-machine` *machine*`)`

Sets the target machine for the subsumption compiler. If there is a Lisp resident retargettable assembler it also makes the assembler's target this machine. The argument *machine* must be a symbol naming a machine. Currently supported machines include

`68k` The 68000, running SOS (Seymour Operating System).

`h6301` The Hitachi 6301. The operating system must be provided by the user as an assembler macro in this case.

`m68hc11` The Motorola 68HC11. The operating system is provided as an assembler macro in this case.

`clsim` Common Lisp. This produces a file of Common Lisp source code that can be compiled by a regular Lisp compiler.

A current machine must be set before the subsumption compiler can be invoked.

*** (`behave` *file &key subcompile listing target*)
Invokes the behavior compiler. The file name defaults to the current directory and an extension of `beh`. For instance: `test.beh` can be specified with a file name argument of `test`. The output file then defaults (and there is no way to change the default) to an extension of `lisp`. So in the previous example the output file would be `test.lisp`. The *subcompile* argument is just a flag. If `nil`, it says to terminate after behavior compiling. If non-`nil` it says to pass on to the subsumption compiler. In that case the `listing` and `target` arguments are passed onto the subsumption compiler. Otherwise they are ignored.

*** (`subcompile` *file &key listing target*)
Invokes the subsumption compiler. The file name defaults to the current directory and an extension of `lisp`. For instance: `test.lisp` can be specified with a file name argument of `test`. The output may be another file with a different extension, or an incore assembly language program, depending on the target machine. For the 68000, an assembly source program is written into a file with extension `asm` (e.g., `test.asm`). For the 6301 and 68hc11 the result is an incore assembly object. A symbol whose print name matches the file name main part (e.g., `TEST` in our example) is created and is bound to this assembly object. In this case the symbol is returned as the result of the procedure call. For Lisp, a file with extension `clisp` is produced.

The *listing* keyword, when non-`nil`, says to create an assembler listing file that contains compiler comments[1]. In the case of the 68000, this means that a normal `asm` file is produced but now comments are included. This option slows the subsumption compiler down by a factor of two but can be very useful for debugging. In the case of machines which don't normally produce an assembly file, the *listing* argument can force the production of one. If *listing* is `T`, then the file name with an extension of `txt` is used. Otherwise *listing* can be a file name itself which will be used directly.

The *target* keyword lets you specify a target machine without having to explicitly set it. If this argument is supplied then a call to `set-current-machine` is done globally, changing the target.

2.2 Examples

In the following two example interactions, exactly the same effects are achieved. In each case the 68000 is chosen as the target machine. The source file `cmor;seymour.beh` is compiled into the intermediate file `cmor;seymour.lisp`. Then an assembly language compilation of that is produced, with comments, in the file `cmor;seymour.asm`.

```
? (set-current-machine '68k)
68K
? (behave "cmor;seymour")

Behavior compiler.
Processing: cmor;seymour.beh
  Processing: cmor;base.beh
  Processing: cmor;basemon.beh
  Processing: cmor;linc.beh
Outputting to: cmor;seymour.lisp

Generated 24 afsms and 11 wire trees.

  29 event-dispatches and 36 dispatch clauses.

T
? (subcompile "cmor;seymour" :listing t)
```

[1]These are comments generated by the compiler explaining what it is doing—they can help an experienced user.

```
Subsumption compiler.  Target: 68K
Processing: cmor;seymour.lisp
Outputting to: cmor;seymour.asm

#.(pathname "cmor;seymour.asm")
?
```

The file seymour includes references to three other files, which is why they are shown getting recursively processed.

The same results can be attained with a single call to behave as below.

```
? (behave "cmor;seymour" :listing t :target '68k :subcompile t)

Behavior compiler.
Processing: cmor;seymour.beh
  Processing: cmor;base.beh
  Processing: cmor;basemon.beh
  Processing: cmor;linc.beh
Outputting to: cmor;seymour.lisp

Generated 24 afsms and 11 wire trees.

  29 event-dispatches and 36 dispatch clauses.


Subsumption compiler.  Target: 68K
Processing: cmor;seymour.lisp
Outputting to: cmor;seymour.asm

T
?
```

In the two examples above, the assembler code was annotated with compiler comments. Here is an example of two pieces produced in the above runs.

```
*           AFSM171:S178 must be suspended
*           test event AFSM171:(DELAY 0.2)
        move.l  d7,d6           ;pick up system clock
        sub.l   regs-24(a6),d6  ;compare to time suspended
        cmpi.l  #200,d6         ;check (delay 0.2)
        blt.s   endmod277
dispatch279
        clr.l   regs-24(a6)     ;unsuspend AFSM171:S178
        bsr     afsm171_s179    ;dispatch
endmod277
.....
afsm171_s179
        clr.b   _BC_tcurrmax(a6) ;primop: CLEAR-BASE-STATUS
        clr.b   _BC_rcurrmax(a6) ;primop: CLEAR-BASE-STATUS
afsm171_s180
        tst.b   regs+7(a6)      ;primop: <
        ble.s   afsm171_s182
afsm171_s181
        moveq   #-5,d2          ;stash output HEADING in temporary location
        or.w    #256,d2         ;set up message arrived flag from HEADING
        move.w  d2,regs+18(a6)  ;  deliver to AFSM188:HEADING
```

The same code when the *listing* argument is `nil` looks like:

```
        move.l  d7,d6
        sub.l   regs-24(a6),d6
        cmpi.l  #200,d6
        blt.s   endmod277
dispatch279
        clr.l   regs-24(a6)
        bsr     afsm171_s179
endmod277
.....
afsm171_s179
        clr.b   _BC_tcurrmax(a6)
        clr.b   _BC_rcurrmax(a6)
afsm171_s180
        tst.b   regs+7(a6)
        ble.s   afsm171_s182
afsm171_s181
        moveq   #-5,d2
        or.w    #256,d2
        move.w  d2,regs+18(a6)
```

3 The Behavior Language

Real-time rules are the key to the behavior language. They can be isolated or grouped into behaviors.

The rules manipulate constants and variable quantities held in registers—usually no more than 8 bits wide.

There are monostables that can be triggered and monitored.

Once a real-time condition is met, some small amount of Lisp-like code gets activated. There are a number of special forms in the behavior language. Their semantics are detailed below, along with the form that expressions can take. There is no notion of procedure definition—all abstraction must be done in macros, rules or behaviors.

At the same time, for complex new robots it is usally necessary to define some new interfaces. This is the role of primops. See section 10 for details.

In most cases the subset of Lisp defined below is downward compatible with Common Lisp.

It is worth noting that programs using real-time rules and calling fragments of Lisp code as defined below have run on processors with as little as 128 bytes of RAM.

3.1 Expressions

Expressions use usual Common Lisp syntax.

An *expression* is either an *arithmetic expression* or the application of a *predicate* to zero or more arithmetic expressions. A predicate is one type of *primitive procedure*—the type that returns true/false.

An arithmetic expression can be a constant (although it does not have to be an arithmetic constant—it could be a string or a keyword for example), or a primitive procedure applied to zero or more arithmetic expressions. In this case, the primitve procedure must return a number (typically in the range $[-128, 127]$; 8 bits signed). Sometimes it is not defined which such number they will return.

Primitive procedures (or primops) are defined in implementation specific ways by the use of code templates. The actual primitive procedures defined may vary between implementations.

At the very least, one can expect every implementation to include: `+`, `-`, `max`, `min`, `=`, `/=`, `<`, `>`, `<=`, `>=`, etc. Individual implementations will document the available primitives in separate documents.

3.2 Logical Expressions

Logical expressions are used as tests in conditional branches. Logical expressions can be either:

- a monostable
- a predicate applied to argument expressions
- (`NOT` *logical-expression*)
- (`AND` `&`*rest logical-expressions*)
- (`OR` `&`*rest logical-expressions*)

3.3 Special Forms

A *special form* is a structure used for flow of control. A special form's evaluation usually includes one or more expression evaluations at some recursive level.

The valid special forms in the behavior language are as follows. Note carefully the places which are referred to as expressions. These really must be expressions as defined in the previous subsection. Otherwise subforms can generally be either expressions or special forms.

*** (`if` *test then* `&`*optional else*)
This is much like the Common Lisp `if`. The only restriction is that *test* must be a logical expression.

*** (`cond` `&`*rest cond-clauses*)
Again, this is like the Common Lisp `cond`. Each clause has the form

(*test* `&`*rest consequents*)

where *test* must be a logical expression (or the special expression `t`) and the consequents can be anything.

*** (`repeat` *(variable range)* &*rest bodyforms*)
This is an iteration construct. *range* must be an expression which evaluates to a positive integer no bigger than 127. The *variable* (it can be named `nil` if it is not referred to in the *bodyforms*) is bound to the *range* value minus one, then stepped down by one to zero. At each bind, (including zero) the *bodyforms* are evaluated. The *bodyforms* can make reference to the *variable* and get the variable's current binding at all times.

*** (`sequence` &*rest forms*)
This is identical to the Common Lisp `progn`. I.e., it provides a sequence construct—hence the name.

*** (`nothing`)
Does nothing. Useful as a place holder sometimes.

*** (`let` *bindings* &*rest bodyforms*)
Just like Common Lisp `let`. Each variable binding has the form:

(*varname valexpression*)

where *valexpression* must be an expression.

*** (`let*` *bindings* &*rest bodyforms*)
Just like Common Lisp `let*`. Each variable binding has the form:

(*varname valexpression*)

where *valexpression* must be an expression.

*** (`setf` *varname expression*)
This is much like the Common Lisp `setf`, or more precisely `setq`. It sets the value of *varname* to the result of evaluating the *expression*.

*** (`trigger` *monostable*)

Triggers the monostable named *monostable*. The monostable must have a time period declared elsewhere. Triggering an already triggered monostable simply elongates the time the monostable is "on" for its full time period from the moment the monostable is most recently triggered.

*** `(output` *portname expression*`)`
This evaluates the *expression* and sends it to the named output port, *portname*. Depending on the context, there is more than one possible meaning for output port. Multiple meanings are explained below, in the section on sharing.

*** `(send` *portspec expression*`)`
This is like `output`, but one specific input port on another entity is named, and the message is sent directly there. This is not such good programming style, as it lets destinations get buried deep inside user code, where a casual observer might miss them. Maybe this should be flushed.

Besides these special forms, `whenever` and `exclusive` can also be seen as special forms when they are not at the top level of a process specification.

4 Specifying AFSMs

An AFSM is built for every *toplevel* real-time rule seen by the behavior compiler. We will specify the meaning of toplevel later, but for now it suffices to include the case of typing a real-time rule as a toplevel s-expression in the input file—i.e. one that is not enclosed in any extra parentheses (list structure).

There are two ways of specifying a real-time rule.

- As a `whenever` form.
- As an `exclusive` form.

The computational model is that for each rule, a single computational process is devoted full time to evaluating it.

4.1 Rule Syntax; `whenever`

The most common way to specify a rule, and hence an AFSM is as a `whenever` clause. The general syntax is:

*** `(whenever` *condition* `&`*rest body-forms*`)`

The semantics of this are that a computational process will be devoted full time to executing this rule. The process starts in a wait state. Whenever the triggering *condition* becomes true, the list of *body-forms* are evaluated sequentially. Then the process returns to the wait state until the condition again becomes true. It is legal for the process to get stuck forever in the body—this can easily happen if there are recursive `whenever` forms, but see below for details.

There are a number of possibilities for the form of a whenever condition. We list them below.

`t` This says that every certain amount of time, the body of the whenever form will be unconditionally evaluated. The *certain amount of time* here, is often referred to as the characteristic time of a particular implementation of the behavior/subsumption system. On all our processors so far, this time has been 0.04 seconds giving a fundamental frequency to the system of 25Hz.[2]

monostable The condition is true for the duration of the triggering of the named monostable.

`(not` *monostable*`)` The condition is true only while the monostable is not triggered.

`(delay` τ`)` This is like the case of `t` above, but now any explicit time period τ can be used to set the frequency of evaluating the body. It should be expressed in units of seconds.

`(received?` *register*`)` This is true if a message has been deposited in the named register since the start of waiting in this `whenever` clause.

`(and|or` `&`*rest forms*`)` This is true if the logical `and` or `or` of the remaining forms are true. The remaining forms can be any of:

- *monostable*
- `(not` *monostable*`)`
- `(delay` τ`)`
- `(received?` *register*`)`

[2]At compile time the Lisp variable `*characteristic-time*` contains the characteristic time.

predicate-form Such an arbitrary lisp predicate (except that it cannot be an *and*, *or*, or *not* form as that would make it ambigous with the case above) is evaluated repetitively at a repeat rate determined by the characteristic time of the implementation. Whenever the *predicate-form* is true, the *body-forms* are evaluated. In essence, using a *predicate-form* is identical to having used:

`(whenever t (if` *predicate-form* `(sequence .` *body-forms*`)))`

4.1.1 Changing the characteristic time

For any particular whenever condition, the characteristic time can be changed by wrapping it in a `with-time` form. The syntax is

`(with-time` *period whenever-condition*`)`

The semantics are simply that the `whenever-condition` is evaluated with a characteristic time specified by the time period.

4.1.2 Non-local exits

It is legal to have a `whenever` form (or `exclusive` form) wherever a special form is allowed. Thus for instance, one might write the following code to make the physical status of a door correspond to the current state of an internal variable.

```
(defconstant $open 0)
(defconstant $closed 1)

(whenever (= message $open)
   (open-the-door)
   (whenever (= message $closed)
      (close-the-door)))
```

There is a problem here however, as once the inner `whenever` is entered, there is no way to exit it. The inner whenever will continually check its condition and perhaps repeatedly initiate the door closing action. In order to break out of such inner `whenever` forms, there is a form named `done-whenever`. For instance:

```
(whenever (= message $open)
   (open-the-door)
   (whenever (= message $closed)
      (close-the-door)
      (done-whenever)))
```

In this case, as soon as the door closing action is initiated, the computational process running this rule reverts to checking the outer condition. Thus, both the door opening and the door closing actions only get initiated once for each change in state of the `message` variable.

The syntax of the form is:

*** `(done-whenever` &*optional (height 0)*`)`

When the optional *height* argument (which defaults to 0) is 0, it means simply to exit the lexically most recent `whenever`. For larger values, it means to exit successively less lexically recent `whenever` forms, each lexical layer traversed decrementing the height by one. E.g.:

```
(whenever (received? mess1)
   (whenever (received? mess2)
      (whenever (received? mess3)
         (print "Got 1, 2, and 3 sequentially")
         (done-whenever 1))))
```

In this example, the process waits for a sequence of messages to arrive in registers `mess1`, `mess2`, and `mess3`, and as soon as the process gets the third it prints the message, then reverts to waiting for a message in register `mess1`. Note that other messages may arrive in the meantime and they are ignored—thus the process will respond to a sequence like $1, 2, 2, 1, 3$ for instance.

4.2 Rule Syntax; `exclusive`

An `exclusive` rule provides a way to simultaneously monitor many conditions and then exclusively service the first one that occurs, ignoring any of the other conditions which might happen during that servicing.

The general syntax is:

*** `(exclusive` &*rest whenever-forms*`)` where *whenever-forms* is a collection of `whenever` rules.

The model is that there is a computational process assigned full time to this `exclusive` rule. This process monitors all the `whenever` conditions simultaneously, and as soon as the first condition happens, the process devotes all its attention to evaluating that `whenever`'s body, temporarily ignoring the other `whenever` conditions. As soon as the body is exited, the process goes back to monitoring all the parallel conditions anew.

The following example distinguishes between messages arriving in an isolated way in the `bar` register, from those that arrive less than or equal to two seconds after a message arrives in register `foo`.

```
(exclusive
   (whenever (received? bar) (print "Isolated BAR"))
   (whenever (received? foo)
      (exclusive
         (whenever (received? bar)
            (print "BAR received within two seconds of FOO"))
         (whenever (delay 2.0) (done-whenever 0)))))
```

Note that if it is ever the case that more than one of the `whenever` conditions becomes true simultaneously, then the leftmost one lexically is the one that is chosen.

5 Behaviors, Machines, etc.

Real-time rules can appear as top-level lisp forms in a source file. As such, they get compiled into single AFSMs, with no user-visible name. Such rules cannot be referred to in order to connect virtual wires to their inputs or outputs. Furthermore, such real-time rules are lexically closed. Any registers these rules refer to are purely local. It is impossible to refer to any monostables as there is no syntax for declaring the monostables' activation time periods within the rules. Therefore, isolated real-time rules at top level are of rather limited value, although they are sometimes useful to initiate background housekeeping processes.

5.1 Single Named Rules

A slightly more useful way of specifying an isolated real-time rule is by giving it a name with `defmachine`. The syntax is:

*** (`defmachine` *name declarations rule*)

The *name* becomes the name of a single AFSM which implements the *rule* specified. The *declarations* slot lets the user specify monostable periods and initial values for registers. The declaration syntax is delineated below.

5.1.1 Registers

Registers as such and output ports are not necessarily declared in the behavior language. Rather their existence can be inferred from seeing references to them. Output port names can be identified syntactically by appearing in an `output` statement. All free references to variables must be either registers or monostables. Monostables must have their time period declared somewhere so they can be disambiguated from regular registers.

Given the name of the AFSM declared in the `defmachine` construct and the implicit output names and inferred register names, the `connect` form described later in this document can be used to connect individual inputs and outputs in so defined augmented finite state machines.

5.1.2 Declaration syntax

The *declarations* slot of a `defmachine` is simply a list of declarations. There are three forms declarations can take:

- (*name* `:init` *value*) which both declares *name* to be a register and gives it an initial value.
- (*name* `:monostable` *period*) which declares *name* to be a monostable and declares its activation period.
- (*name* `:additive (`*l h*`))` which says that all incoming messages to this register should be *added* and the sum should be bounded in the range $[l, h]$. No overflow checking of intermediate values is done in these computations, although the bounding computation is done after each message is sent to such a register.

For instance consider a machine using 8 bit signed arithemetic with an additive register with the range `(0 120)`. Suppose its value is `120` and a message of `15` arrives. The sum will be `135` which overflows the capacity of an 8 bit machine, and this will confuse the max instruction done. Thus, users must beware of the possible size of the arguments that will be added.

Any register not declared by an `:init` has an initial value that is undefined.

Multiple declarations on a single register can simply be appended as a set of keywords and values.

5.2 Collections of Rules: Behaviors

Collections of real-time rules can be grouped into behaviors. There are a number of advantages to such grouping:

1. Registers, monostables and outputs can be shared across many real-time rules.
2. Behaviors provide a coarse scale abstraction barrier which even strong temptation cannot broach.
3. Fewer names are needed in the name space.
4. Complete sets of rules (i.e., behaviors) can be activated through multiple activation mechanisms.

There are two ways to specify behaviors; with `definterface` and with `defbehavior`. For now we will consider these two forms to be indentical. Later we will distinguish them.

The form of a behavior specification is:

*** `(defbehavior` *name* `&`*key inputs outputs decls processes*`)`

or equivalently

*** `(definterface` *name* `&`*key inputs outputs decls processes*`)`

The arguments are as follows:

name The name of the behavior.

inputs A list of registers which are available as inputs from outside the behavior.

outputs A list of output ports which can be connected to external entities.

decls A list of declarations having the same syntax as those in `defmachine`.

processes A list of real-time rules, having the same syntax as those described earlier. These rules can be expressed as `whenever`, `exclusive`, or even `defmachine` statements. In the latter case the rule has a somewhat useless name.

As with single-rule machines, any undeclared free variable in a behavior description is assumed to be a register. Any name appearing in an `output` statement is assumed to be an output. However, unless a name appears in the *outputs* specification list, it cannot be exported from the behavior. If the name also appears elsewhere as a free variable, it is given dual treatment described below and handled as a register also.

5.2.1 Sharing within behaviors

All registers and monostables, with the exception of those that also appear on the input list, or as output names somewhere within a rule, are shared across all the AFSMs generated for the behavior. We will treat registers which appear as outputs separately in the next section.

Consider the following example behavior:

```
(defbehavior tester
  :inputs (f1 f2)
  :decls ((total :init 0))
  :processes ((whenever (received? f1)
                 (setf total (+ total f1)))
              (whenever (received? f2)
                 (setf total (- total f2)))))
```

Here the register `total` is shared between two AFSMs, one of which increments it, while the other decrements it.

Likewise, monostables can be shared between rules, and when one rule triggers a monostable the other rule will see it as having been triggered.

When two rules in the same behavior refer to a name they are referring to the same entity.

There are two exceptions to this sharing concept; registers that are named explicitly as inputs and registers which accept internal sends. The latter case is discussed in the next subsection. Registers that are named explicity as inputs are replicated in each AFSM where they are referenced. This is so that the `(received? ...)` construct will work in each rule independently. Thus, different rules within a behavior will refer to different copies, and the values will diverge if one rule includes a `setf` of the appropriate name.

5.2.2 Message passing within behaviors

It is possible to pass messages between rules within a single behavior. This mechanism was referred to above has having internal sends.

If there is an output to a port name that is the same name as is used in syntactic positions reserved for registers, then such output messages are sent to each copy of that register—one copy for each real-time rule that refers to it. It is not necessary in this case that the port name also appear as a declared output of the behavior, although it may, and in that case all messages are also sent out along any virtual wires connected to that port.

The reason for having internal message passing rather than just shared registers, is to enable synchronization between rules—that is exactly what happens when one rule is waiting to trigger on a `received?` clause when a message from another rule arrives.

5.2.3 Behaviors and interfaces

There are two differences between behaviors and interfaces (specified with `defbehavior` and `definterface` respectively).

The primary intent of the differentiation is that interfaces should implement virtual sensors and actuators.

Note that this is different from saying that interfaces are used to differentiate central from peripheral systems. Any sensor is a virtual sensor at some level—the time taken for a sonar return to arrive is represented on one particular wire as a direct physical analog by the time taken for a voltage to go high, whereas a little further down the processing line the analogy is stretched a little as it becomes data on a 16 element wide binary bus. Interfaces in the behavior language simply push that abstraction another step until the sensor readings and actuator commands become virtual at the level of appearing as messages on connections between sets of real-time rules. There is no necessary sense here in which perception becomes peripheral to the central system. Perception still may be done in and at the behest of behaviors. The distinction between interfaces and behaviors is that behaviors deal only with messages on connections and internal state.

The implementation of this primary intent is through enforcement of the primitive procedures that may be used in rule sets specified with `defbehavior`. Certain primitives are designated as communicating, either from sensors or to actuators, outside the realm of behavior language entities. Their use is forbidden except in interfaces.

A secondary, and indeed subsidiary, intent is that behaviors but not interfaces can be activated and deactivated as complete units. There are two schemes for such behavior control, described below in the activation section. These schemes are accessed by additional keywords to `defbehavior` that are not legal for `definterface`.

6 Connections

The method for connecting isolated AFSMs and behaviors together is the `connect` form. Isolated AFSMs and complete behaviors can be mixed and matched within such declarations of virtual wires. Isolated AFSMs and behaviors are treated equivalently.

A source or destination of a wire is written as a port identifier. The general form for such a thing is:

(*objectname portname*)

where *objectname* is the name of either an explicitly constructed isolated AFSM, or the name of a behavior. The *portname* is either an implicit register or output in the case of an AFSM specified with `defmachine`, or in the case of a behavior it is an explicitly declared input or output.

6.1 Explicit Wires

The general form for explicit connection is:

*** (`connect` *source dest1* &*rest more-dests*)

This says that for every output at the *source* port, a copy will be delivered to every destination specified as *dest1* or in the list *more-dests*. If a destination port name is part of a behavior, then a copy of the message gets delivered to every rule or explicit AFSM that references the named input.

Connects can also be used to implement suppression and inhibition. In this case the destination is either

((`suppress` *input-port*))

or

((`inhibit` *output-port*))

or

((`default` *input-port*))

In each case *input-port* and *output-port* take the form of port identifiers decribed earlier.

The sematics of suppression are that the new connection sends its messages to the old *input-port*. When such a message is sent, it completely blocks any messages for the old port from other sources for some time period—twice the characteristic time of the implementation.

The semantics of inhibition are that the new connection inhibits any outputs getting out of the old *output-port* for some time period after an inhibiting message is sent. That time period is again twice the characteristic time of the implementation.

The semantics of default are just like those for suppression, except that it is the old wire that has dominance over the new wire.

6.2 Implicit Wires

Wires can be implicitly built into a rule specification, by using the `send` special form. It looks like:

*** (`send` *destination value*)

In this case no declared output is used. In fact, this syntax is internally used to transform the source behavior so that it has an explicit new output port name. The `send` form is transformed into a syntactically identical `output` form, and an explicit `connect` form is added.

Thus:

```
(defbehavior foo
   :processes ((whenever (with-time 1.0 t)
                   (if (check-clock)
                       (output bar 33)))))
```

```
(connect (foo bar) (some place))
```

is completely equivalent to:

```
(defbehavior foo
   :processes ((whenever (with-time 1.0 t)
                  (if (check-clock)
                      (send (some place) 33)))))
```

7 Macros of All Flavors

Common Lisp-like macros can be defined at the top level of a file by using `defmacro`. Such definitions are actually treated using the underlying Common Lisp mechanism so all the syntax and semantics of Common Lisp apply.

Macros can be used anywhere within the behavior language. They will be expanded appropriately. They can, of course, expand into other macro calls and all will be handled appropriately. Note however, that only macros which occur within behavior language source files will be expanded. Any lisp macro which already happens to be in the environment will be ignored.

7.1 Top Level Macros

It may be convenient to have a top level macro return many items, for instance a couple of behaviors and some connections between them. To enable this to happen, there is a special form which can occur only at top level, or recursively nested within itself, named `collection`. It takes the form:

*** (`collection` `&`*rest forms*)

The *forms* are treated exactly as if they themselves had occured at toplevel of the source file (and hence they too can include a `collection`).

7.2 Units

A special restricted form of macro that is easy to define is used to declare units of measurement for any constant. This is necessary because in most implementations of the behavior language/subsumption architecture, numerical quantities (apart from time periods) are restricted to be 8 bit signed integers—i.e., they range from -128 to 127 inclusive.

The general form is:

*** `(defunit` *unitname (arg) form*`)`

The idea is that the *form* provides a mapping from quantities in the named units into the range $[-128, 127]$. For instance representing degrees ranging from $[-360, 360]$ in the range $[-120, 120]$, we could define:

```
(defunit degrees (x) (round x 3))
```

Then we could refer in behavior code to quantities like `(degrees 270)` which would translate into 90, small enough to fit into 8 bit registers.

8 Activation Mechanisms

Behaviors can have two states; an *active state* and an *inactive* state. The rules that make up a behavior can actually be segmented into three classes that operate differently depending on the state of the behavior. The classes are specified using additional keywords. The rule specification keywords for behaviors are:

`:processes` These rules always operate as specified earlier.

`:h-processes` These *haltable processes* cannot run at all when the behavior is in its inactive state. It is as if every `whenever` condition has an additional term that is not satisfied.[3]

`:i-processes` These *inhibitible processes* always run, but all outputs are inhibited while the behavior is inactive.

There is a monostable called `active-p` which is available to any rule. The `active-p` monostable says whether the behavior is active or not. Of course, it makes no sense to access this monostable from a haltable rule, since whenever a haltable rule is running it must be the case that `active-p` is triggered.

[3]And indeed that is how it is implemented!

8.1 Activation

There are four keyword slots to `defbehavior` concerned with describing when a behavior is active. These are:

`:precondition` This is an aribtrary expression which should return true or false. When it is true it triggers a user visible monostable `preconition-p` with a period of twice the characteristic time. This monostable must be activated before a behavior can be activated, as below. This slot is optional and can be omitted. In that case it is as though the precondition is always true.

`:activation` This is an arithmetic expression which is evaluated at a frequency determined by the characteristic time of the implementation. Typically it is evaluated at 25Hz. The expression in this slot can refer to any register within the behavior. Other possibilities are described in subsequent subsections. The evaluation of the expression results in a number in the range $[-128, 127]$. This is user visible as the contents of the pseudo register `activation-level`.

`:threshold` This is a number in the range $[-128, 127]$ which is compared to the computed activation level of the behavior. If that activation level is greater than or equal to the threshold and, when a `:precondition` was satisfied, if the `precondition-p` monostable is active, then the `active-p` monostable is triggered. The behavior is active if and only if the `active-p` monostable is on.

`:continuance` By having a monostable in addition to the regular comparison with the threshold, some hysterisis and stability is built into the system. This slot specifies the time period of the monostable. If the slot is omitted the time period defaults to 2.0 seconds.

8.2 The Hormone System

The hormone system is a crude form of a behavior activation mechanism.

There are two types of non-procedural entities:

`condition` These are named excitation quantities which decay over time (see below) but which can be excited by any process which chooses to excite them. A condition's value ranges from 0 to 15, although there is no explicit way for this value to be examined.

`releaser` These are functions of the conditions. Releaser values are kept up to date by background processes, with a time lag of no more than the characteristic time of the implementation.

Together, conditions and releasers form a low bandwidth global communication mechanism.

A coarse analogy is that conditions are emotions and releasers are hormones.

The forms for defining conditions and releasers are `defcondition` and `defreleaser`. Their general forms are:

*** `(defcondition` *name* `&`*key ...*`)`

*** `(defreleaser` *name* `&`*key ...*`)`

In both cases, *name* is a symbolic name for the quantity. Both forms take a number of optional keywords listed below.

For conditions, there is an initial value declared (see below). That value can be increased at any time (up to the maximum value of 15) with the `excite` primitive which takes the form:

*** `(excite` *condition-name* `&`*optional (amount 1)*`)`

`excite` can be used in the body of any real-time rule in the whole system. There is no lexicographic requirement. The named condition is incremented by the desired amount (defaulting to 1).

The value of each condition decays over time. The decay rate can be determined by the user. It defaults to a linear rate of one unit every 12 seconds, but it can be changed, and also made bi-linear. The decay mechanism is quite general, but the underlying assumption in the syntax for specifying it, is that in the bi-linear case there is a region of hyper-activation where the decay rate is lower.

The keywords for `defcondition`, along with their default values are:

`init 0` The initial value for the condition.

`decay-period 12` The number of seconds between a reduction of one unit of the condition level.

`hyper-level nil` If this is non-`nil` then at this level or above, a slower rate of decay is used.

`hyperize 24` The number of seconds between an *increase* of one unit of the condition level when in the hyper active region.

There is only one keyword for `defreleaser`.

`generation 0` An expression on constants and previously declared conditions. This expression gets evaluated at a rate determined by the characteristic time of the implementation in order to keep the releaser up to date.

Any releaser can be referred to in the `:activation` slot of a behavior. In this way, the hormone system can activate selected behaviors.

8.3 Spreading of Activation

Besides hormones, there is a direct method for spreading of activation between behaviors. The current direct method has some drawbacks and will be modified in the future after experience is gained with it. The direct method will also be modified to allow the implementation of certain ideas in learning.

The principal idea behind activation (from [**Maes 89**]) is that any behavior can spread some portion of its activation to other behaviors. This spreading is modelled as a continuous process (although it is implemented discretely). Spreading activation does not diminish the current activation level of a behavior.

As before, the activation level of a behavior is specified by the expression in the `:activation` slot. Besides registers and releasers the expression can also refer to the virtual register `received-activation`. The contents of the `received-activation` register is determined by activation messages (ideally thought of as continuous values on wires, although it is possible to abuse this model) from other behaviors. There is a primitive named `direct-activate`. It takes the form:

*** `(direct-activate` *behavior-name portion*`)`

Here, *behavior-name* is the name of some other behavior. The result of evaluating this form is to add some activation to the `received-activation` register of that other behavior. The amount received is the sending behavior's activation multiplied by the *portion*. The *portion* must be a rational number with a denominator which is a power of two (since some of our target machines have no divide instruction!).

The `received-activation` virtual register, by default, has values in the range $[0, 63]$ where overflow and underflow are banged against these limits. This range can be altered with the `:received-activation-range` keyword argument to `defbehavior`. It should look like a list of the low value and high value. The range must include 0.

Although spreading of activation should be thought of as a continous thing it is implemented discretely by having `received-activation` be an `:additive` register. Every time the `:activation` expression is evaluated (at the characteristic frequency of the implementation) this register is effectively reset to 0. Thus the correct way to use `direct-activate` is to run it at the characteristic frequency, most typically as a `whenever t` rule.

9 Arrays

An array is a one dimensional vector of registers, each of which acts like an individual register—e.g., a message arriving at an array element can be used to trigger a `whenever` rule, things can be stored in the array, and elements can be read and used. Note that an array can be an input to a behavior—in effect a vector of inputs. An array can also be a one dimensional vector of outputs.

Like registers, all arrays are statically allocated at compile time. For arrays this means that their size must be declared as a constant. An array is declared in the `:decls` slot. An array declaration has two possible forms; one without element initialization and one with all elements being initialized to the same constant. The forms of these declarations are:

- (*name* `:array` *size*) which declares an array called *name* to have the constant *size* number of elements.
- (*name* `:array` *size* `:init` *value*) which additionally declares that all elements of the array have initial value *value*.

A declared array can also be listed as in input or as an output.

The primitive for accessing components of an array is `aref`.

*** (`aref` *arrayname index*)
Arrays are indexed with a zero base. An `aref` form can appear anywhere a regular register or output portname can appear (assuming the array is declared appropriately). In particular it can be a location to which a `setf` refers.

When an `aref` is used in a `connect` statement as either an input portname or an output portname the value of the index must be computable at compile time. Thus, there cannot be any parameterized wires joining behaviors.

When an `aref` is used as the destination of an `output` statement the value of the index must also be computable at compile time. Thus, there cannot be any output redirected dynamically [4].

An `aref` form can appear in a `received?` test condition of a `whenever` clause even when the index is not computable at compile time. Thus, input can be redirected dynamically. Note, however, that unlike regular registers, arrays that are inputs are not duplicated over multiple rules, so that there is only one chance to poll the flag that says the message has actually arrived.

```
;;; we use come-from registers rather than goto registers because
;;; an output must be to a known location at compile time.  but we
;;; can use simple indexing to select from a behavior's input registers.
;;;
;;; the following switch handles 8 inputs and 8 outputs.  each output
;;; can have at most one input, and only the first attempt at checking
;;; an input will notice a message had arrived.
;;;
;;; to request a switch setting, send an input index to inpsel, an output
;;; index to outsel, and a 1 to each of inp-count and out-count.  the
;;; latter two are for collision detection.  the status output tells whether
;;; there was a collision or not and does nothing if there was.  after
;;; a switch has been set, all messages get routed appropriately.

(defconstant $switch-success 1)
(defconstant $switch-failure 0)

(defmacro passon (index)
  `(whenever (received? (aref a (aref b-i ,index)))
             (output (aref b ,index) (aref a (aref b-i ,index)))))

(defbehavior switch8
```

[4]The implementation reason for this is that it would require making the various suppressed and inhibited wire trees connected to a particular output be callable as a subroutine. The original BL implementation was not set up in a way which makes this easy.

```
  :inputs (a inpsel inp-count outsel out-count)
  :outputs (b status)
  :decls ((b-i :array 8 :init 0)
          (a :array 8)
          (b :array 8)
          (inp-count :additive (0 63) :init 0)
          (out-count :additive (0 63) :init 0))
  :processes ((whenever (and (received? inpsel)
                             (received? outsel))
                        (cond ((and (= inp-count 1)
                                    (= out-count 1))
                               (setf (aref b-i outsel) inpsel)
                               (output status $switch-success))
                              (t (output status $switch-failure)))
                        (setf inp-count 0)
                        (setf out-count 0))
              (passon 0)
              (passon 1)
              (passon 2)
              (passon 3)
              (passon 4)
              (passon 5)
              (passon 6)
              (passon 7)))

;;; now for a silly behavior to demonstrate how connections are made

(defbehavior foo
  :inputs (baz)
  :outputs (bar)
  :processes ((whenever (received? baz) (output bar baz))))

(connect (foo bar) (switch8 (aref a 3)))

(connect (switch8 (aref b 5)) (foo baz))
```

10 Connecting To The Outside

The behavior language is only useful if it can do something in the world. In this section we discuss how to make the actual conenctions to the world.

10.1 Simulation Using Common Lisp

A simple way to make connections to the world is to use a simulation of it. This can be most easily done in the behavior language by setting the current machine to `clsim`[5].

The behavior language compiles subsequent input files into Common Lisp source code in files with an extension of `.clisp`. This file can be compiled (by the Common Lisp compiler) and loaded. Then to run the behavior program use the form `(scheduler)`.

Any lisp procedure can be called by a behavior language program, but all free variables will be treated as register names.

*** `(lisp` *form*`)`
This treats `form` as vanilla lisp and does not try to look through it for references to registers. Instead this piece of lisp code will be compiled and called directly at the appropriate place.

10.2 Reserving storage

For backends that produce code that run directly on other machines it is sometimes necessary to allocate storage which is not behavior language registers. This can be done with:

*** `(defdata` *label size*`)`
which allocates *size* bytes of data and gives a symbolic *label* to the zeroth one. This label may be referred to inside primitive operators (see below) or with the two forms:

*** `(examine` *'label*`)`

[5]If the CL simulator is not already loaded, you can use the form `(load-clsim)` to get it in.

*** `(deposit` *'label value*`)`
These take a quoted label and either examine its contents or set its contents. In reality these two operations are implemented as primitve operators (see below).

10.3 Initializing Other Processes

There is a way to escape to assembly language to make specific calls during initialization. This is achieved through the form:

*** `(initialize-forms` &*rest forms*`)`
Each item in the list *forms* should be an assembly language instruction to be run during initialization. Typically it will be a subroutine call to the operating system.

It is legal to have multiple `initialize-forms` spread throughout a behavior language program. They are collected together automatically sequentially in the order of detection.

10.4 Primops

Primops are code templates for procedures called within behavior language programs. When a user wants to extend the interface between the behavior language and the underlying hardware the typical thing would be to write a new primop. The primops should be compiled and loaded in the lisp environment in which the behavior compiler runs. In `clsim` all lisp procedures are treated as primops, so there is no need for this section in that case.

A primop has the form:

*** `(defprim` *name* &*io-p args extra result isrep code opt coll*`)`

The idea is that this provides a procedure at compile time which, when given some operands specifying the location of arguments, will generate appropriate code, and also return an operand describing where the result of the procedure can be found. Note that these operands are machine operands that machine instructions can refer to.

`name` The name of the procedure—it can be called by this name from within the behavior language.

`io-p` A flag (`nil` or `t`) which says (in the latter case) that this is an I/O procedure. Such procedures can only be invoked from interfaces and not from regular behaviors.

`args` A list of argument specifications, giving the operands symbolic names, and placing any restrictions on legal classes of operands. It is up to the compiler to find a way to deliver the arguments in operands of the appropriate classes. Each element of this list is a list of two things; an argument name, and an operand class.

`extra` A list of extra operands that this primop would like to be able to use (e.g., some extra registers). This list takes the same format as the argument specifications.

`isrep` This is either `value` or `branch`. The former says that this procedure returns an actual value, the latter says that it sets a condition code which can be branched on.

`result` In the case of an `isrep` of `value`, this is an operand describing where the result of the procedure will show up. Note that this is an expression which gets evaluated. It can be a backquoted form, for example. Any named locations in the `args` and `extra` slots can be used as free variables in this form. In the case of `isrep` of `branch` then this is a condition name (e.g., `eq`, `ge`, etc.,) which is what to branch on in the true case.

`code` This is an expression which should evaluate to a list of instructions. Typically it will be a backquote form which will make reference to the operands specified in `args` and `extra`.

`opt` A form which can refer to the free variable `form`, which will be bound to the expression calling the named procedure. If this form returns a different form that the original source at compile time, then that new source is used instead. This is used for example, to turn addition of many arguments into a series of additions of two arguments.

`coll` When specified this should be a procedure (e.g, `#'*`) which when applied to all literal arguments gives the result that the procedure being defined would have given at runtime. This is used by the compiler to eliminate procedure calls with all literal arguments.

The operand classes are machine dependent. On the 68000 the legal classes are `dreg`, meaning a data register, and `any` meaning an operand. On the 6301 and 6811 the legal classes are `!accum-a` meaning the A accumulator and `anyna` meaning any non-accumulator operand (i.e., constant or memory). In both cases arguments can also declared to be `literal` which means that at compile time the value must be able to be determined, and that will be what the argument name is bound to, when evaluating the `code` slot.

Consider the following examples from the 6301:

```
(defprim oddp
  :io-p ()
  :args ((x !accum-a))
  :result ne
  :isrep branch
  :coll #'oddp
  :code `((anda (! 1))))

(defprim max
  :io-p ()
  :args ((x anyna) (y !accum-a))
  :result y
  :isrep value
  :code (let ((endlabel (gentemp "MAX")))
          `((cmpa ,x)
            ((branch ge) ,endlabel)
            (ldaa ,x)
            ,endlabel))
  :coll #'max
  :opt (dyadicize-form form))

(defprim examine
  :io-p t
  :args ((p literal))
  :extra ((x !accum-a))
  :result x
  :isrep value
  :code `((ldaa ,p)))
```

The `oddp` example is a non-value producing primop. It sets condition codes, and a subsequent branch operation will be affected by that condition code. In the example, the argument comes in the A accumulator, which is anded with 1. If the result is non-zero that means the argument must have been odd, so checking for the condition `ne` ensures that the branch will be taken on the true case. The compiler is free to optimize this and may decide to check for the negation of the condition, i.e., `eq` and branch on falseness of odd.

The second example, `max`, must build a more complex piece of code with an branch and labels.

The third example illustrates the use of a literal argument, in this case it is an assembler label, possibly reserved with `defdata`.

11 Summary of Allowed Top Level Forms

The legal toplevel forms in a behavior source file are:

- `defconstant`
- `defmacro`
- `defmachine`
- `definterface`
- `defbehavior`
- `defunit`
- `defcondition`
- `defreleaser`
- `connect`
- `whenever`
- `exlusive`
- `collection`

- `defdata`
- `initialize-forms`
- A macro defined in the current compilation context using `defmacro`

12 Acknowledgements

Pattie Maes provided much input and influence over the design of the activation spreading mechanism. She and Maja Mataric have been the primary guinea pigs subjected to use of the behavior language. Pattie and Anita Flynn had many useful comments on earlier drafts of this document.

13 References

[**Brooks 86**] "A Robust Layered Control System for a Mobile Robot", Rodney A. Brooks, *IEEE Journal of Robotics and Automation*, RA-2, April, 14–23.

[**Brooks 89**] "A Robot that Walks: Emergent Behavior from a Carefully Evolved Network", Rodney A. Brooks, *Neural Computation 1:2*, Summer.

[**Connell 89**] "A Colony Architecture for an Artificial Creature", Jonathan H. Connell, *MIT Ph.D. Thesis in Electrical Engineering and Computer Science*, June.

[**Maes 89**] "The Dynamics of Action Selection", Pattie Maes, *AAAI Spring Symposium on AI Limited Rationality*, IJCAI, Detroit, MI, 991—997.