Functional Programming of Behavior-Based Systems

Ian Douglas Horswill

Computer Science Department and The Institute for the Learning Sciences
Northwestern University
Evanston, IL 60201
ian@cs.nwu.edu

Abstract

In this paper, I describe a simple functional programming language, *GRL*, in which most of the characteristic features of the popular behavior-based robot architectures can be concisely written as reusable software abstractions. This makes it easier to write clear, modular code, to "mix and match" arbitration mechanisms, and to experiment with variations on existing mechanisms. I describe the compilation process for the language, our experiences with it, and issues of efficiency, expressiveness, and code size relative to other languages.[1]

Introduction

In this paper I describe a simple language, *GRL* (Generic Robot Language, formerly "GiRL", and still pronounced "girl"), that extends traditional functional programming techniques to behavior-based systems.[2] *GRL* is an architecture-neutral language embedded within Scheme. It provides a wide range of constructs for defining data flow within communicating parallel control loops and manipulating them at compile time using functional programming techniques. A simple compiler (about 3000 lines of code) then distills the code into a single while-loop containing straight-line code, and emits it as Scheme, C, C++, BASIC, or Unrealscript[3] code. The language allows programmers to write in a much more modular and compositional manner, but generates C code that is typically faster than hand-written C code.[4] We have implemented the language and used it as the basis for our behavior-based systems at NWU for approximately two years. We have run it on a half-dozen robot configurations ranging from a sonar-based system with an 8-bit microcontroller to a vision-based quadruped. It has also been used to teach an undergraduate seminar.

Although evaluation of programming languages is always difficult, students who have used both raw LISP and *GRL* report that *GRL* code is much easier to write and debug. As an anecdotal example, the NWU entry in the 1998 AAAI robotics exhibition was a program that stalked people in the crowd and followed them around while spinning paranoid fantasies about the government turning people into robots. It was constructed in three weeks by one of our undergraduates. It was his first robot program.

Programming languages and behavior-based systems

In spite of its name, GRL is really only intended for implementing behavior-based systems (it's called *GRL* because "GBBL" would be hard to pronounce). Although many different behavior-based architectures exist (see Arkin 98 for a survey), all consist of a set of independent control loops running in parallel, usually with some sort of arbitration mechanism to combine their outputs. Many of the distinctions between behavior-based architectures amount to disagreements about what a behavior is and how a system should arbitrate between behaviors. In subsumption (Brooks 86), a behavior is a finite-state machine and arbitration is performed with suppression. In the motor-schema architecture (Arkin 98), behaviors are mappings from sensor vectors to motor vectors, and arbitration is a performed using linear combination. In behavior networks (Maes 90), behaviors (competence modules) are unrestricted black boxes, but are arbitrated using a spreading activation mechanism that forms an approximation to STRIPS planning (Fikes, Hart, Nilsson 72).

Many behavior-based architectures are implemented as application-specific programming languages and compilers. For example, the MIT mobot lab has developed a series of languages for writting subsumption

[1] Support for this work was provided in part by the National Science Foundation under award #IRI-9625041 and in part by the Defense Advanced Research Projects Agency Mobile Autonomous Robot Software Program under award N66001-99-1-8919 and by the DARPA Distributed Robotics Program and the U.S. Army Soldier Systems Command under award #DAAN02-98-C-4023/MOD P3.

[2] The name is, of course, a gross exaggeration. It was contrived to allow the name.

[3] Unrealscript is the extension language for a popular computer game.

[4] This is achieved in part because the *GRL* compiler is maniacal about inlining, so its object code, while very fast, is typically somewhat bloated. This could be changed, but so far, it hasn't been a problem.

code (Brooks 86, Brooks 90, Brooks and Rosenberg 95). Maes' paper describes her architecture, in part, in terms of a macrology and interpreter embedded in LISP. Thrun (98) describes a C-like language that supports probability distributions and gradient descent function approximators as primitive data types. Several researchers have also written architectures as C++ class libraries (Velásquez 98, Schaad 98). Unfortunately, since most robotics researchers don't have time to be professional compiler writers, these systems are often quite primitive as programming languages. In particular, their abstraction capabilities, if any, are typically limited to macro expansion. They also tend to make it painful or impossible to implement the other architectures as user code. Thus one can't easily compare the subsumption architecture with behavior-networks because they have completely separate languages. It is possible of course, but it typically involves hand-compiling one architecture into the primitives of another.

The goal of *GRL* is to have a language with the abstraction facilities of LISP and the run-time efficiency of the subsumption compiler. Although it is limited to the kinds of real-time finite state computations used in behavior-based systems (and therefore cannot implement full symbolic programming systems like theorem provers), it does provide the programmer with an architecture-neutral language for experimenting with different kinds of arbitration mechanisms.

GRL primitives

GRL supports *circuit semantics* (Nilsson 1994). Ultimately, *GRL* programs consist of networks of signals which are thought of as being computed in parallel and updated continuously. In reality, of course, *GRL* code runs on serial uniprocessors or loosely coupled networks of uniprocessors, so the parallelism and continous update are only approximate. Any signal that exists at run time must be a *primitive* signal, meaning more or less that it must be easily computed in C. A primitive signal may be:

- A *constant*
- A *signal source*, for which raw Scheme code has been provided by the programmer. They are used for sensor and effector interface code.
- An application of a *primitve procedure* (+, -, log, …) to a set of signals. Note that `if`, `and`, and `or`, are normal procedures in *GRL*, not special forms.[1]
- An application of a *finite-state transducer* to a set of signals. Again, the programmer must provide Scheme source code for computing the value of the transducer from the instantaneous values of the input signals. Transducers are finite state in that they may not dynamically allocate storage.

Primitive procedures are mappings from instantaneous signal *values to values*. Values may be scalars (integers, floats, Booleans, …) or vectors. Transducers, since they contain history information, are mappings from signal *time histories* to *time histories*. However, they are restricted to be finite state. Sources and transducers are the only constructs in which programmers are allowed to escape to Scheme and write Scheme code that will be executed at run time. This Scheme code is restricted, however, to be statically typeable and to be non-consing, at least if the compiler is to generate C or BASIC code for the final output. Assuming the code provided for sources and transducers runs in $O(1)$ time, then the network as a whole runs in $O(1)$ space and can compute updates in $O(1)$ time.

Primitive signals make up a straightforward language for writing real-time control loops. Although most of the advantages of *GRL* come from more sophisticated constructs, the primitive signal sub-language does have the advantage that allows functions over time histories (transducers) to be composed more naturally than is possible in most programming languages. For example, consider the problem of detecting a doorway using a directional proximity sensor, such as a sonar or IR, while driving down a hallway. A simple approach is to look for a long reading from one of the sideways-facing sensors:

```
(define-signal door?
  (> (max left-reading
          right-reading)
     thresh))
```

However, this might generate spurious readings because of sensor errors. Programmers often post-process the sensor output by requiring that the long reading be seen for at least a certain amount of time:

```
(define-signal door?
   (> (true-time
       (> (max left-reading
               right-reading)
          dist-thresh))
      time-thresh))
```

[1] This is for technical reasons. Conditionals are usually special-forms in languages with applicative-order evaluation because programmers typically don't want conditionals to pre-evaluate all their subexpressions. Since GRL signals have quasi-parallel semantics, conditionals always evaluate all their arguments. Care must be taken, therefore to ensure that unintentional range errors do not occur, such as in the expression `(if (zero? a) 0 (/ 1 a))`.

where `true-time` is a finite-state transducer that returns the number of milliseconds for which its input has been true. It is a standard part of the *GRL* library and is defined by the code:

```
(define-transducer (true-time input)
  (state-variables (onset 0))
  (when (not input)
    (set! onset ms-clock))
  (- ms-clock onset))
```

Unfortunately, you can't write `true-time` conveniently in LISP or C because it doesn't behave like a normal function – it requires an internal state variable, `onset`. Although both LISP and C allow functions to have internal state (for example with C static variables), they only provide a single set of those variables. If `true-time` were called anywhere else in the program, the single `onset` variable would be forced to do double-duty with disastrous results. While there are workarounds for this (lexical closures or C++ class objects), they require explicit instantiation and binding of separate instances of `true-time` for each use, which is ugly and hard to maintain. In practice, programmers tend to inline the code manually. This leads to a proliferation of explicit side effects and variables with names like `door-onset`, `stuck-onset`, `door-other-onset`, *etc*. The resulting code is error-prone and hard to read. *GRL* allows `true-time`, and similar transforms such as `one-shot`, `integral`, `derivative`, and `hysteresis`, to applied and composed in a natural, functional style.

Procedural abstraction

GRL supports compile-time procedural abstractions called *signal procedures*. Signal procedures are mappings from signal networks to signal networks. Since they run at compile time, they are allowed to use the full power of Scheme. In this sense, they're very similar to LISP macros. We distinguish them from macros because (1) signal procedures are evaluated in applicative order (arguments first, then procedure body), rather than normal order, as with macros and lazy evaluation, and (2) signal procedures are not applied to source text, but to the actual signal networks. Signal procedures therefore cannot quote their own arguments, return non-syntactic results, or shadow variable bindings. *GRL* does support the underlying Scheme macro system, however, so macros can be used in cases where implicit quoting or argument destructuring is desired.

Compile-time data abstraction

GRL supports data abstraction through *compile-time* list and record structures. Record structures are typically defined using the define-group-type macro which defines constructor and accessor functions for the data type:

```
(define-group-type xy-vector
  (xy-vector x y)
  (x x-of)
  (y y-of))
```

This example declares that the xy-vector data type[1] consists of `x` and `y` elements, which are accessed with the `x-of` and `y-of` procedures, respectively. New xy-vectors are created with the `xy-vector` procedure, which takes the x and y components as arguments.

Groups are explicitly represented at compile-time so that they can be manipulated and simplified. The compiler can thereby avoid generating run-time code for accessors, i.e. the compiler reduces the expression `(x-of (xy-vector c d))` to the expression `c`. It can also avoid generating any code whatsoever for group components that are never accessed.

GRL also supports compile-time lists. These are used principally to allow signal procedures to accept variable numbers of arguments. Again, as with groups, all calls to list-related functions are resolved at compile time, so that no list structures are ever created or manipulated at run time.

Implicit mapping

To reiterate, primitive procedures map values to values, transducers map time histories to time histories, and signal procedures map circuits to circuits. Primitive procedures and transducers, as such, are not defined over lists and groups. However, a natural extension of their semantics is to define them to *map themselves* over the elements of the list or group. This has the effect of automatically extending arithmetic operators to vectors in the standard manner, so that:

```
(+ (* 2 (xy-vector a b))
   (* 4.5 (xy-vector c d)))
```

is reduced by the compiler to the expression:

```
(xy-vector (+ (* 2 a)
              (* 4.5 c))
           (+ (* 2 b)
              (* 4.5 d)))
```

Vector and array operations

GRL supports vectors and arrays as run-time data types. Like groups, most primitives are implicitly mapped over the elements of vectors (there are a few exceptions, like

[1] *GRL* also supports real arrays, which also support implicit mapping. However, vectors are often defined as groups so that their elements can be given meaningful names. Otherwise, it can be hard to distinguish between, e.g. polar vectors and Cartesian vectors.

`vector-length`, that would make no sense to map), although transducers are not mapped over vectors in the current version. This often allows relatively complex looping operations to be expressed very concisely. For example, discrete sampling of a continuous function can be written compactly as the following signal procedure:

```
(define-signal (sample f sampling)
  (f (sampling-index->value
       sampling
       (index-generator
         (sampling-samples sampling)))))
```

Here, a `sampling` is a group type that represents how a function is to be sampled – the interval and the number of samples. Index-generator is the equivalent of the APL ι operator. (index-generator 3) returns the vector (0 1 2), (index-generator 5) returns the vector (0 1 2 3 4), etc. `Sampling-index->value` then maps these integer values to the values at which the function `f` is to be sampled, and then `f` is then mapped along the resulting vector to obtain the vector of sampled values.

Given `sample`, we can then write a potential field obstacle avoidance algorithm for sonar-based robots as:

```
(define-signal (avoid sonar-dists)
  (vector-sum
    (* (sample (xy-vector sin cos)
               (sampling 0 (* 2 pi) 16))
       (/ 1 sonar-dists))))
```

The call to `sample` computes a vector of unit xy-vectors in the directions of the different sonars (we assume sonar ring with 16 sensors). These vectors are then multiplied by the forces generated by the sonar readings to create a vector of force vectors. Finally, this vector is summed to compute an aggregate force vector.

While this algorithm may appear inefficient, the compiler is actually quite aggressive about inlining and constant folding. The resulting code, after cleaning up to be more readable, will look approximately like:

```
float sumx;
float sumy;
float costable[16] = { 1.0, … };
float sintable[16] = { 0.0, … };

sumx = 0;
sumy = 0;
for (i = 0;  i<16; i++) {
  sumx = sumx+costable[i]/sonardists[i];
  sumy = sumy+sintable[i]/sonardists[i];
}
```

Notice that the sampled unit vector has been constant folded to a pair of lookup tables.

Other features

Although space precludes a full discussion, the *GRL* language also includes other useful such as:

- *Accumulators*. These are signals whose inputs are left unspecified at the time they are defined. The programmer can then explicitly declare other signals to drive those accumulators. The compiler will then fill in the inputs of the accumulator at compile time with the set of signals within the current compilation that have been declared to drive the accumulator.
- *Macros*. For extending the syntax of the language.
- *Modalities*. These are signal procedures whose values are only default values. They can be overridden on a signal-by-signal basis by the programmer.
- *Operator overloading*. At present, this only has limited support, but we intend to extend it.
- *Support for symbolic algebra*. Signal procedures can examine the operator, inputs, and type of each input they receive. This makes it possible to write signal procedures that perform symbolic differentiation, integration *etc*.
- *Compile-time property lists*. These have turned out to be less useful than expected.

Compilation

In large part, the compiler looks like a partial evaluator and type inference system that operates as a front-end to a system like Rex (Kaelbling 87) or the Subsumption compiler (Brooks, unpublished). Roughly, the compiler runs by computing the signal graph and simplifying it as much as possible. Then it performs standard optimizations, such as inlining, algebraic simplification, constant folding, and hoisting of loop invariants.

The compiler is called with a list of signals to compile. It runs in several passes:

1. *Signal expansion*. The compiler recursively evaluates signal procedure applications, group constructors and selectors, and applications of primitive procedures to lists or groups, until it has reduced the program to a graph of primitive signals. Note that this guarantees that signal procedures are always called with networks of primitive signals as their inputs, provided the network is acyclic.
2. *Topology determination*. The compiler computes the set of input signals for each accumulator.
3. *Topological sort*. The compiler performs a postorder traversal of the primitive signal network. The result is a total ordering of the primitive signals that drive the original signals passed to the compiler. This ordering is used to determine the order of evaluation

in the object code.

It is possible to specify recurrence relations by creating cyclic signal graphs, e.g. using `letrec`, the recursive version of `let`. Cycles in the graph of primitive signals are "broken" by inserting a unit delay before one of the signals in the cycle, called the "cycle breaker." The value computed in the last time-step for the cycle breaker is always used during the current time-step to compute the values of other signals. If one of the signals is declared to have an initial value, then it will always be chosen as the cycle breaker. If no signal is declared with an initial value, the compiler will issue a warning message, make an arbitrary choice, and assume a default initial value for the signal (0 in the case of numeric signals).

Note that the set of top-level signals passed to the compiler are used as roots for the topological sort, so group elements that are never used are discarded in this step. No object code is generated for them.

4. *Analysis.* This consists of a series of optimization passes on the ordered signals:
   1. *Type inference*. The data types of the values of the signals are determined based on their operators, inputs, and optional declarations.
   2. *Loop-invariant detection.* Signals that are either constant or the result of applying primitives to other invariant signals are flagged. These signals are computed once at boot time.
   3. *Dependency analysis.* For each signal, the compiler determines the set of signals for which it is an input.
   4. *Control structure grouping.* The compiler scans the set of signals for conditionals and loops that share the same test or iteration count. It groups these signals so that each set will generate a single conditional or loop in the object code. This saves a considerable amount of conditional branching at run-time.
   5. *Inline selection.* The compiler tags signals that are either constants, loop invariants, or complex expressions that are referenced exactly once. Uninlined signals are stored in global variables at run time. Inlined signals are incorporated directly into the intermediate code of their dependents during reification (see below). References to individual elements of inlined vector-valued signals are also inlined.
6. *Reification.* The compiler generates Scheme code equivalent to each primitive signal. These expressions are guaranteed to be statically typeable, have no functional arguments, and perform no dynamic storage allocation.
7. *Intermediate code generation.* The compiler constructs a set of variable declarations, and a while loop containing the reified forms of all uninlined signals.
8. *Partial evaluation.* The compiler then performs a number of symbolic simplifications of the intermediate code, such as algebraic simplification of sums and products, inline expansion of min, max, arg-min, and arg-max, and reductions in strength (e.g. replacing a modulus operation with a bitwise-and operation). When necessary, it also massages the code to be more palatable to C and BASIC compilers.
9. *Final code generation.* The compiler then pretty-prints the intermediate code in the desired target language.

Arbitration schemes as higher-order functions

We now present several examples of *GRL* code that show how common popular behavior-based programming mechanisms can be written as normal user-level abstractions in *GRL*. This allows programmers to adopt a mix-and-match approach to robot programming, rather than having to commit to a specific arbitration mechanism. It also makes it very convenient to experiment with variant and hybrid arbitration mechanisms. Please note that space precludes the inclusion of sufficient code to express a real robot program. As a result, the examples are **extremely compressed** and **extremely contrived**.

Motor Schema combination

In motor schema systems (Arkin 98), behaviors generate motor vectors that are combined through weighted summation. To model motor schemas, we first represent behaviors as data abstractions:

```
(define-group-type behavior
  (behavior activation-level motor-vector)
  (activation-level activation-level)
  (motor-vector motor-vector))
```

Next, we define the weighted sum operator, using the activation levels as weights:

```
(define-signal (weighted-sum . behaviors)
  (apply +
         (weighted-motor-vector
behaviors)))

(define-signal (weighted-motor-vector beh)
  (* (activation-level beh)
     (motor-vector beh)))
```

This fragment requires some explanation. First, the dot notation in `weighted-sum`'s argument list means that `behaviors` is a rest arg – it gets bound to a list whose elements are the actual parameters of the call to `weighted-sum`. When `weighted-motor-vector` is called on this list, it expands to the `*` expression in its body. `Activation-level` and `motor-vector` are both signal procedures that expand to `select` calls. These are automatically mapped over the list, so their results are a list of signals, and a list of groups, respectively. The compiler then maps the `*` call over the lists. However, since the elements of the second (motor-vector) list are themselves groups, it again maps over the group elements. The result is then a list of groups, each of which is the product of a vector and a scalar. When the `+` procedure is then `apply`'ed to the list of groups, it is again mapped over the group elements to produce as a result a single group which is a linear combination of the original behaviors' motor-vectors.

As a simple example, we will combine a behavior that homes in on a target with another behavior that avoids the nearest obstacle. To simplify the exposition, we will assume that the perceptual system delivers Cartesian coordinates for the robot, its goal, and the nearest obstacle via the signals `my-position`, `goal-position`, and `obstacle-position`, respectively. We can then define the behaviors themselves:

```
(define-signal move-toward-goal
  (behavior 1.0
            (-      goal-position      my-
position)))

(define-signal avoid-obstacles
  (let ((force (- my-position
                  obstacle-position)))
    (behavior (/ 10
                 (magnitude-squared
force))
              force)))
```

using the ancillary definitions:

```
(define-signal (square x)
  (* x x))

(define-signal (magnitude-squared v)
  (+ (square (select x v))
     (square (select y v))))
```

Finally, we combine them:

```
(define-signal motor-output
  (weighted-sum move-toward-goal
                avoid-obstacles))
```

and compile the signal `motor-output`. The compiler procedures the intermediate code:

```
(set! force-x (- my-position-x
                 obstacle—position-x))
(set! force-y (- my-position-y
                 obstacle-position-y))
(set! avoid-obstacles-activation-level
      (/ 10 (+ (* force-x force-x)
               (* force-y force-y))))
(set! motor-output-x
      (+ (- goal-position-x my-position-x)
         (* avoid-obstacles-activation-level
            force-x)))
(set! motor-output-y
      (+ (- goal-position-y my-position-y)
         (* avoid-obstacles-activation-level
            force-y)))
```

This code can be executed efficiently in Scheme without cons'ing. However, the compiler can also emit the code as C, C++, BASIC, or Unrealscript code.

Now suppose we decide that rather than using a weighted sum, we would prefer a weighted *average*. We simply add the definitions:

```
(define-signal (weighted-average
                  . behaviors)
  (/ (apply weighted-sum behaviors)
     (apply + (activation-level
                behaviors))))

(define-signal motor-output
  (weighted-average move-toward-goal
                   avoid-obstacles))
```

and recompile.

Sequencing

Behavior-based systems typically consist of a collection of simple computations running in parallel. These computations are usually cheap compared to a CPU context switch. As a result, compilers such as Brooks' subsumption compiler (Brooks 86) produce a round-robin schedule for the tasks at compile time and inline them inside a single large loop. When a task requires sequential control, a state variable is added to record which step of the sequence the task is executing. The compiler then wraps all steps of the task in a large case statement. While this seems inefficient and inelegant at first blush, it is between one and three orders of magnitude more efficient than the full context switch required to implement the sequence as a separate thread, depending on the underlying CPU architecture and operating system. It also means the compiler can easily generate code for low-end microcontrollers whose operating systems (if any) do not support multithreading.

Although *GRL* provides no direct support for sequencing, it is simple to write it as a user-level extension. Suppose we want to write a sequencing construct that, given a series of output signals and a series of condition signals, relays the first output signal until the first condition signal becomes true, then moves on to the second output signal, *etc*. We also probably want to include an input that resets it to the beginning of the sequence. We could write this in *GRL* as a signal procedure:

```
(define-signal (sequence-proc reset?
                              output-signals
                              conditions)
  (letrec
    ((step (counter done-w/step? reset?))
     (done-w/step?
       (list-ref conditions step))
     (output
       (list-ref output-signals step)))
    output))
```

Here `letrec` is the standard recursive form of let, which is extended in *GRL* to allow the definition of mutually recursive signals. `Counter` is a transducer that counts from zero, incrementing each time its first input is true. It resets to zero when its second input is true. `List-ref` is used here to build a multiplexer: it takes as arguments an list of signals and an index *step* into the list returns as a result the value of the *step*'th signal. It compiles into a case statement.

Unfortunately, the sequencing construct above requires that the programmer provide the lists of outputs and termination conditions separately, which is inelegant, to say the least. We would prefer to group output/condition pairs into clauses, as in `cond` statements. This requires writing sequence as a macro, however:

```
(define-signal-syntax sequence
  (syntax-rules ()
    ((sequence (reset? first-output)
               (condition output) ...)
     (signal-expression
      (sequence-proc reset?
                     (list first-output
                           output …)
                     (list condition
…))))))
```

Space precludes a discussion of the pattern matcher for the Scheme macro system (see (Rees et al.) for a discussion). However, an example should make the idea clear:

```
(define-signal motor-output
  (sequence
    (start?     follow-freespace)
    (left-turn? follow-freespace)
    (left-turn? stop)))
```

This generates a state machine that drives forward (or more properly, outputs the output of the follow-freespace behavior) until it reaches the second left-hand turn, at which point it stops.

While state machines are often good sequencers for navigation, it is often desirable to allow the sequencer to move *backward*, should one of the previously achieved conditions suddenly become false. Such a sequencer is called a *teleo-reactive tree* (Nilsson 94). A TRT always executes the leftmost action whose termination condition is unsatisfied. TRTs are implemented using nested if-then-else structures and can also be generated with a simple macro:

```
(define-signal-syntax trt-sequence
  (syntax-rules ()
    ((trt-sequence ?val)
     ?val)
    ((trt-sequence  ?val1 (?term ?val2)
                    ?stuff ...)
     (signal-expression
      (if ?term
          (trt-sequence ?val2 ?stuff ...)
          ?val1)))))
```

Behavior competition

Another common arbitration mechanism is competition, in which the most strongly activated behavior is chosen. We can write this as a higher order procedure:

```
(define-signal (behavior-max . behaviors)
  (list-ref behaviors
            (apply arg-max
                   (activation-level
                     behaviors))))
```

Remember that `behaviors` is bound to a list of behaviors passed in as arguments. The `activation-level` call then computes the list of their activation levels (since groups and lists are resolved at compile time, this doesn't actually involve any computation at run-time). The call to `arg-max` then finds the position in the list of the highest activation behavior, and, finally, the call to `list-ref` returns that.

Spreading activation

Another popular type of behavioral competition is *spreading activation*, in which the activation levels of the different behaviors are computed as differential equations of one another (e.g. Maes 89, Velásquez 98). In *GRL*, this is can be written as an application of `letrec` and some kind of temporal filter, typically a low-pass filter:

```
(letrec ((a (low-pass-filter (- a-input b)
                             a-decay))
         (b (low-pass-filter (- b-input a)
                             b-decay))
   (behavior-max (behavior a a-output)
                 (behavior b b-output)))
```

This will dynamically choose between `a-output` and `b-output` based on the activation levels `a` and `b`, which are `mutually inhibitory and` stimulated by inputs `a-input` and `b-input`, respectively. However, we might prefer to be able to write:

```
(spread-activation
  (a :output a-output
     :input a-input
     :inhibitors (b)
     :halflife a-halflife)
  (b :output b-output
     :input b-input
     :inhibitors (a)
     :halflife b-halflife))
```

This is easily done with another macro, however space precludes its inclusion here. A control structure such as Maes' behavior networks (1991), while not trivial, is a straightforward extension.

Implementation and Evaluation

The *GRL* language has been in use at NWU for approximately two years. It has been used on a number of platforms ranging from a differential-drive sonar-based machine controlled by an 8-bit microcontroller to prototypes of the Sony *Aibo* pet robots. Among other things, it was used to implement a person-follower demonstrated at AAAI-98. The user community of *GRL* presently includes approximately 10 researchers, a few undergraduates using it for outside projects, and a two interns from a local high school. It is also used to teach the autonomous robot courses at NWU.

Evaluation of programming languages is always difficult. Since programming languages are algorithm-neutral, the choice of language doesn't affect robot performance. However, the choice of programming language does affect programmer productivity and code maintainability. Often, the choice of programming language comes down to aesthetic issues, which are necessarily subjective.

The main advantage of *GRL* is that it allows the programmer to write implementations of higher level operators such as subsume, TRTs, and spreading activation, and then call them in a clean, compositional style. While we believe this makes the code cleaner and easier to maintain, there is no way to quantify this. Students in an undergraduate course on behavior-based robotics reported that *GRL* code was easier to write and debug than raw LISP code, but we have not performed any controlled studies of this.

What we can quantify is that *GRL* code is very concise. We have implemented a number of popular behavior-based programming features and measured their code sizes. The results are given in the following table:

Feature	**Lines of code**
Motor schemas	18
Brooks subsume operator	7
Tinbergen lateral inhibition	11
Spreading activation	8
Modal Horn clause inference (compiles limited first-order Horn clauses with modalities into a TMS-like network)	144
Propositional production system	48
Society-of-Mind-like frame system	88
Teleo-reactive trees	9
FSM-based sequencing	9
First-order low-pass filter	8
Hysteresis thresholding	9

By comparison, the raw LISP implementation of the frame system was 151 lines of code. The modal Horn-clause inference system (Horswill 98) was 583 lines of LISP code with another 337 lines written in C for speed. Both systems had fewer features than their *GRL*-based counterparts and ran two orders of magnitude slower, since they were interpreters rather than compilers. In fairness to LISP, one should note however that since *GRL* only supports symbolic manipulations at compile time, it would be impossible to implement an interpreter in it even if one wanted to. It should also be noted that the LISP system in question was quite primitive.

Related work

Behavior-based robotics researchers have traditionally avoided complex control and data structures, in part because of the need for efficiency. There have been notable exceptions however. Kaelbling's REX system (1986) was also a language that allowed programmers to lay out synchronous circuits using LISP programs. REX allowed programmers to generate representations of circuits using LISP programs, then compile those representations to 68000 assembly code. The compile-time language for building circuits did not attempt to mimic functional programming semantics, although it was Turing complete, so programmers could in principle write higher-order functions in it. Nilsson's TRT system

(1994) used a LISP interpreter that incrementally grew a circuit at run-time, much the way a rule-based inference engine might cache its results in a TMS. This gave the programmer LISP-like control structures, including full unbounded recursion, at the cost of additional run-time overhead. Brooks and Rosenberg's L system (1995) provides a reasonably full version of Common LISP that's suitable for use in embedded systems, together with a macrology for writing subsumption-style control systems.

More recently, a number of C-like languages for robot programming have been developed. Schaad (1998) developed a uniform architecture called *parallel functional decision trees* that was able to emulate many of the popular behavior-based programming constructs, including both parallel and serial control structures. Although his system was implemented as a C++ class library, he also provided an interpreter that built the PFDT network at run time from a LISP program. Thrun's CES language (1998) extends C with probabilistic data types and support for gradient descent function approximators. A particularly appealing property of Thrun's language is that training information need not be specified at the output of the function approximator. Instead, desired values can be provided for other variables that are computed from the output of the approximator. The compiler does the appropriate symbolic manipulations to compute the gradient of the derived quantity with respect to the weights of the approximator. Simmons and Apfelbaum (1998) describe an extension to C++ called TDL that provides support for subgoal decomposition, coroutining, and synchronization. They have also implemented a graphical front-end for the language that allows a form of visual programming. MacKenzie and Arkin (1997) describe another visual programming tool, MissionLab, however it is intended to allow novice programmers to efficiently task robots, not to program new low-level behaviors. Konolige's COLBERT (1997) is another C-like language for low-level reactive systems, that is designed to interface with the Sapphira architecture (Konolige et al. 97).

There have also been a number attempts to apply modern programming language technology to robot programming. Rees and Donald (1992) describe a minimalist multithreaded Scheme interpreter that ran in 100K on a 68008 microprocessor. They used the system to control sonar-based mobile robots. The system was very successful, however speed issues made it more useful for sequencing than for implementing high-frequency control loops.

Recently, Peterson et al. (1999) have described an embedded language in Haskell called Frob, which can be used for reactive control loops and is very similar in spirit to our work. The most important functional difference between Frob and GRL is that Frob is considerably more expressive than GRL (since it includes all of Haskell, even at run time). The downside of this is that it requires the overhead of a full Haskell interpreter, whereas our work has focused on building the compiler technology necessary to generate code for low-end microcontrollers that is competitive with hand-written C code. Frob has been used for a number of real-time robot control applications, including visual tracking.

Finally, Levesque et al. (1997) have developed a logic programming language called GOLOG that combines the automatic inference capabilities of logic programming with the explicit control operations of imperative languages. Like Frob, it provides considerably more expressiveness than GRL at the cost of increased execution time. GOLOG is intended more for planning and problem solving than for implementing low-level control, however, so this is hardly surprising.

Conclusion

Many aspects of behavior-based architecture, particularly arbitration mechanisms, are best thought of as higher order operators that map behaviors to more complicated behaviors. A good behavior-based programming language should be powerful enough to allow the programmer to experiment with different kinds of operators. They should be able to mix and match the popular ones, as well as experiment with new ones.

Traditional functional programming techniques provide an elegant substrate for constructing and manipulating behavior-based control systems. The *GRL* language is an example of a language allowing full-use of mapping, reduction, and higher-order functions at compile-time, while still generating code that is competitive with, or better than, hand-written C code. The compiler is simple (about 3100 lines of Scheme code) and is highly retargetable, allowing the same code to be run on everything from high-end research robots to low-end microcontrollers.

Most importantly, *GRL* is largely architecture neutral. It imposes minimal commitments on the programmer, requiring only that object code consist of a network of signals and communicating finite-state machines (although out-calls to other programming models, such as planners are supported).

The major design consideration in *GRL* was to make it as easy as possible for programmers to implement other people's architectures. The examples given in this paper, while contrived from a robotics standpoint, are

intended to demonstrate how the prevailing programming models can be concisely written as short macros or higher-order procedures. This allows programmers to mix-and-match different architectural primitives and to experiment with new ones.

The GRL system can be obtained on the web from http://www.cs.nwu.edu/groups/amrg/distributions/grl.

Acknowledgements

I would like to thank Mark DePristo, Robin Hunicke, Aaron Khoo, Dac Le, Olin Shivers, Pinku Surana, Robert Zubek, and the anonymous reviewers of this paper for their generous comments. I would also like to thank Richard Kelsey and Jonathan Rees for building the Scheme48 system, NEC Research for supporting it, and Lars Bergstrom for keeping our Windows port of it running. Finally, I would especially like to thank Aaron Khoo for sending me lots of bug reports, and my wife Louise for being tolerant when I fixed those bugs while we were watching T.V. together.

References

Arkin, A. 1998. *Behavior-Based Robotics*. MIT Press, Cambridge, MA.

Brooks, R. 1986. "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, pp. 14-23.

Brooks, R. 1990. "The Behavior Language," *A.I. Memo No. 1227*, MIT AI Laboratory, April.

Brooks, R. and Rosenberg, C. 1995. "L – A Common LISP for Embedded Systems," *LISP Users and Vendors Conference*, sec 2.4a.

Fikes, R, Hart, P., and Nilsson, N. 1972. "Learning and Executing Generalized Robot Plans," *Artificial Intelligence* Vol. 3, No. 4, pp. 251-288.

Horswill, I. 1998. "Grounding Mundane Inference in Perception," *Autonomous Robots*, Vol. 5, pp. 63-77.

Kaelbling, L. 1987. "Rex: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems," *Proceedings of the AIAA Conference on Computers in Aeorospace VI*, Wakefield, MA, pp. 255-60.

Kaelbling, L. and Rosenschein, S. 1991. "Action and Planning in Embedded Agents," in *Designing Autonomous Agents*, ed. P. Maes, MIT Press, Cambridge, MA, pp. 35-48.

Konolige, K. 1997. "COLBERT: A Language for Reactive Control in Sapphira" In *Proceedings of the German Conference on Artificial Intelligence*, Freiberg.

Konolige, K., K. Meyers, A. Saffiotti, and E. Ruspini, 1997. The Saphira architecture: a design for autonomy, *Journal of Experimental and Theoretical Artificial Intelligence*, **9** (1997) pp. 215-235.

Levesque, H., R. Reiter, Y. Lespérance, F. Lin, and R. Scherl 1997. "GOLOG: A logic programming language for dynamic domains". *Journal of Logic Programming*, **31**, 59-84, 1997.

MacKenzie, D. and R. Arkin 1997. *Evaluating the Usability of Robot Programming Toolsets*. Technical Report, Georgia Institute of Technology. Atlanta, GA, October 1997.

Maes, P. 1989. "The Dynamics of Action Selection," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, pp. 991-997.

Maes, P. 1990. "Situated Agents Can Have Goals," *Robotics and Autonomous Systems*, Vol. 6, pp. 49-70.

Nilsson, N. 1994. "Teleo-Reactive Programs for Agent Control," *Journal of Artificial Intelligence Research*, Vol. 1, pp. 139-158.

Peterson, J. and G. Hager and P. Hudak 1999. "A Language for Declarative Robot Programming" In *Proceedings of the 1999 International Conference on Robotics and Automation*, Detroit, MI, May 1999. IEEE Press.

Rees, J. and B. Donald 1992. "Program Mobile Robots in Scheme." In *Proceedings of the IEEE International Conference on Robotics and Automation*. Nice, France (May 1992), pp. 2681-2688.

Schaad, R. 1998. *Representation and Execution of Situated Action Sequences*. Dissertation Der Wirtschaftswissenschaftlichen, Universität Zürich.

Simmons, R. and D. Apfelbaum 1998. "A Task Description Language for Robot Control." In *Proceedings of the Conference on Intelligent Robots and Systems*. IEEE Press, October, 1998.

Thrun, S. 1998. *A Framework for Programming Embedded Systems: Initial Design and Results*.

Technical Report CMU-CS-98-142, Carnegie Mellon University, Pittsburgh, PA.

Velásquez, J. 1998. "When Robots Weep: Emotional Memories and Decision-Making," *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, Madison, WI, pp. 70-75.