FROB: A Transformational Approach to the Design of Robot Software

Gregory D. Hager[1] **and John Peterson**[2]

[1] Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, hager@cs.jhu.edu

[2] Department of Computer Science, Yale University, New Haven, CT 06520, peterson-john@cs.yale.edu

Abstract

Over the last two years, we have applied methodologies developed for *domain-specific embedded languages* to create a high-level robot control language called *Frob*, for *F*unctional *Rob*otics. The goal of Frob is to provide an environment where robot software can be clearly, cleanly, and correctly specified, while suppressing unnecessary implementation detail. To this end, Frob incorporates three important notions: an implicit representation of time, equational specification of continuous-time behaviors and asynchronous events, and functions to construct complex, reusable abstractions. We show that these ideas can be combined to produce many commonly used "higher-level" abstractions for specifying robot programs. Experience with this approach to robot programming suggests that it leads to effective development of robot software while producing programs that are readable and can be reasoned about in a formal sense.

1. Introduction

In this paper, we present *Frob* (for *F*unctional *Rob*otics), a unified framework for rapidly and reliably creating robotic software ranging from simple behavioral loops to large systems involving complex control strategies and/or multiple interacting modules.

Frob is largely motivated by two observations:

- To a large degree, modern robotic devices are no longer constrained by hardware limitations or computing power — in fact their capabilities are largely limited by our ability to produce the software needed to correctly couple perception to action.
- The (as yet) strongly experimental nature of robotic systems engineering places a strong emphasis on *rapid prototyping* and *software reusability* [1, 2].

To put it another way, experience suggests that effective robot software construction is an incremental process of development, testing, and refinement. Thus, a good development environment should allow a user to pull standard tools (e.g. wall followers, control algorithms, and so forth) "off the shelf" and to customize those tools to a given application, both at the level of basic functionality, and at the level of systems architecture. It should be easy to test the resulting software system against a wide variety of operating conditions with minimal debugging of component algorithms.

The approach we have chosen is based on DSEL (Domain-Specific Embedded Language) technology [3]. The idea of a DSEL is to use a general-purpose language customized to deal with a specific domain of interest. One advantage of this approach is that, in addition to the basic vocabulary we have built within Frob, we can create new abstractions using a fully-featured functional programming language (the Haskell language [4]). As a result, Frob functions at all levels of abstraction, from simple expressions to patterns that capture system architectures.

In the remainder of this paper, we first introduce the basic concepts of Frob with particular emphasis on its ability to abstract over time-varying phenomena. We then describe how these basic structures can be used to develop the notion of a *task*, a more powerful and intuitive programming structure for robotic systems. We illustrate how, through task transformation, a library of basic tasks can be customized to a given application. Finally we show how tasks can in turn be used to develop other well-known software architectures, and close with a discussion of our experience to date with the system.

All code and type signatures presented in this paper are in Haskell. Readers unfamiliar with Haskell should consult `http://haskell.org`. We also invite the reader to consult `http://haskell.org/frob` for full details of the Frob implementation and more extensive examples.

2. Behaviors and Equational Specification

Frob uses two essential abstractions to describe values that vary over time: *behaviors* and *events*. These form the basis for a programming style known more broadly as Functional Reactive Programming (FRP), a set of primitive operations that define the basic interplay among behaviors and events [5]. In this section, we briefly describe how events and behaviors allow for straightforward equational specification of robot behavior.

2.1. Continuous Behaviors

Behaviors are an abstraction of functions over continuous time. Behaviors describe values that are continuously changing, such as the current position of a mobile robot in the plane. Formally, the robot position would be represented as function from continuous time to points in $\Re^2$. Using behaviors, this type is written as `Behavior Point2`. The type expression `Behavior Point2` is part of the type language of Frob (and Haskell); it denotes the application of a *polymorphic* type, `Behavior`, to a constituent type, `Point2`. Thus, `Behavior` serves as a function over types. This form of polymorphism is similar to that used to describe arrays in more traditional languages.

Overloading allows typical operators, $e.g.$ +, to be "lifted" to transparently deal with continuous as well as static information. Behaviors also support novel operators — for example, given a behavior b, `derivative` b is also a real-valued behavior defined in Frob.

To illustrate how behaviors can be used, consider a simple feedback control algorithm[1] for wall following using range (e.g. sonar) data. The system to be controlled is a differential drive robot with control inputs v and ω representing the desired translational and rotational velocity of the robot, respectively. The algorithm makes use of the readings of the front and side range sensors, f and s, as well as the current velocity v_{curr} of the robot. In equations, we write

$$\begin{aligned} v &= \sigma(v_{\max}, f - d^*) \\ \omega &= \sigma(\sin(\theta_{\max}) * v_{\text{curr}}, s - d^*) - \dot{s} \end{aligned}$$

where $\sigma(x, y)$ is the limiting function $\sigma(x, y) = \max(-x, \min(x, y))$, d^* is the desired "setpoint" distance for objects to the front or side of the robot, and $v_{\max}$ and $\theta_{\max}$ are the maximum robot velocity and body angle to the wall, respectively. The strategy expressed here is fairly simple: the robot travels at its maximum velocity until blocked in front, at which time it slows down as it approaches the obstacle. It turns toward or away from the wall based on its distance relative to d^*, but at no time is the side range sensor allowed to be more that θ_{max} degrees away from the perpendicular to the wall.

These equations are rendered in Frob as follows:

```
wallFollow v_curr s f d_star =
 (v, omega)
   where
    v      = limit v_max (f - d_star)
    omega  = rerror - derivative s
    rerror = limit (v_curr * sin theta_max)
                   (d_star - s)

limit high x = max (-high) (min x high)
theta_max = 10 # degrees
vmax      = 50 # cm_per_sec
```

Even to someone who has never seen Frob, the correspondence between this program and the equations above should be very clear. In particular, note that time is *implicit* in the function, just as it was in the original set of equations. The result returned by `wallFollow` has type `WheelControl`, which is a pair of floating-point behaviors, the first being the robot speed and the second being the turning rate. This type is defined by:

```
type WheelControl =
  (Behavior Float, Behavior Float)
```

[To help in reading subsequent Frob programs, here are a few comments on syntax: Juxtaposition is used to denote function application, which always binds tighter than infix operations. Thus `vel * sin thetamax` corresponds to `vel*sin(thetamax)` in more traditional languages. This also applies to definitions: `add x y = x + y` instead of `add(x,y)=x+y`. The notation `(` e_1 `,` e_2 `)` denotes a pairing of the values of e_1 and e_2. The type of a function is given using ::. For example, `f :: a -> b -> c` says that f is a function that produces something of type `c` when supplied with values of type `a` and `b`. Functions can be partially evaluated and can be passed as data objects, so `f:: (a -> b) -> c` is also a legitimate type. It expresses a function that accepts a function from `a`'s to `b`'s and produces thereby something of type `c`.]

To run this program we need to define a mapping from robot sensors to robot effectors; Frob defines the following

```
sonar    :: Robot -> Int -> Behavior Float
velocity :: Robot -> Behavior Float
runScout :: (Robot -> WheelControl) -> IO ()
```

The type `Robot` is a Haskell structure that packages all of the information on a robot (currently Nomadic

[1] Here and elsewhere we have, in the interest of brevity, suppressed algorithm details ($e.g.$ gain coefficients) which are not essential to our presentation of Frob.

SuperScout II's). The `sonar` function selects an individual sonar behavior from the robot; sonar 0 and 3 are the front and right side respectively. The `velocity` function returns the behavior that is the current velocity of the robot. Finally `runScout` accepts a function which produces `WheelControl` when given a robot, and performs the IO actions required to implement the control.

Putting this all together, we can "package" the wall follower with the appropriate sonar for following a wall on the right by writing:

```
frontsonar r = sonar r 0
rightsonar r = sonar r 3

wallfollow1 d_star r =
    wallFollow (velocity r)
       (frontsonar r) (right-
sonar r) d_star
```

We can then execute right-wall following at a distance of 20 centimeters by writing:

```
runScout (wallfollow1 20)
```

Note that, since `wallFollow` is defined on behaviors, there is no loop to iteratively sample the sensors, compute parameters, update control registers, etc. In general, details pertaining to the flow of time are hidden from the programmer. Some operators, notably `derivative` in this example, directly exploit the time-varying nature of the signals. Finally, note that the code is independent of the kind of sensors used to measure the distances or, more generally, how it is derived. For example, we could easily compose `wallFollow` with a function that performs filtering to clean up the incoming sonar data.

2.2. Events and Reactions

Not all values are best represented in a continuous or clocked manner. For example, the robot bumpers, low-battery warnings, keyboard presses, and so forth generate discrete, asynchronous events rather than behaviors. This motivates Frob's notion of an *event*.

As with behaviors, a polymorphic data type defines events: `Event` γ. As events are asynchronous in time, they cannot support the same sort of arithmetic combination used with behaviors — adding two numeric event streams is not meaningful since the time of the occurrences in the two event streams may not match. On the other hand, two event streams can be merged using the `.|.` operation. Events may also be synthesized from behaviors, using the `predicate` primitive. For example, `predicate (s > r)` is an event which occurs when the behavior `s` exceeds the value of the behavior `r`.

Events direct the course of behaviors via *reaction*. The `untilB` function defines a behavior which reacts to an event defining a new behavior. For example, this function:

```
goAhead r t =
  forward 30 `untilB`
      (predicate (time > t)
          .|.
       predicate (frontSonar r < 20))
        -=> stop)
```

can be read, for robot `r`: "Move forward at a velocity of 30 cm/sec, until either time exceeds `t` sec, or an object ahead appears closer than 20 cm, at which point stop." The `untilB` changes the system behavior from `forward 50` to `stop` in response to an event. The back-quotes around `untilB` designate it as an infix operator. The `-=>` operator couples an event with response. In the following section, we generalize this pattern using *tasks*.

3. Tasks and Transformational Programming

Complex systems are often described in terms of a succession of operating modes. While mode transitions can be described using the `untilB` operator, we instead choose to build our systems at a higher level of abstraction using *tasks*. Simply put, a task defines a behavior that terminates on an event and returns a value. We note the notion of a task is closely related to that of a *process* as defined in [6, 7]; it may also be thought of as a discrete unit of a hybrid system or similar finite state machine [8].

Due to space limitations, the description of tasks is necessarily brief; we refer the reader to [9] for more detail on tasks and their implementation in terms of the underlying FRP operators such as `untilB`.

This implementation of tasks is not in any way built in to Frob. Rather, tasks have been constructed using the basic tools of the underlying Haskell system, functions and types, and could be defined by the user instead of the developers of Frob. The ability to create new abstractions such as tasks is the essential contribution of Frob. Thus, no single architectural framework has been wired in to Frob. Users may create new high level abstractions such as tasks to fit their specific requirements.

3.1. Basic Tasks

Generically, tasks are a pairing of a continuous behavior and a terminating event. More formally, the type `Task b e` describes a task that defines a continuous behavior `b` and is terminated by an event generating a value of type `e`.

Tasks can be generated in a variety of ways. For the purposes of this paper, we will generate them *transformationally* starting from two orthogonal primitive tasks. The first primitive is the *empty task* produced by `nullTask`:

```
nullTask :: e -> Task b e
```

Empty tasks execute instantaneously, immediately returning a specified value. The second primitive is the never-ending task produced by `liftB`:

```
liftB :: Behavior b -> Task b e
```

The `liftB` function simply "lifts" a behavior into a task. Since the task created by `liftB` never ends, there is no need to specify a precise type for the returned value: `e` may be any type. Similarly, the type `b` in `nullTask b e` is also arbitrary: an empty task does not determine the type of the continuous behavior.

The essential operation on tasks is sequential composition. Sequential composition is implemented using `do` notation, a feature of Haskell. We write

```
do { t1 ; t2}
```

to sequence tasks t1 and t2. Information dependency between the tasks is handled by the `<-` operator, which binds the returned value (the generated event) of the first task into a variable which is scoped over the second task:

```
do { r <- t1;
     t2 r }
```

For example, by using `liftB`, we can "package" the wall following behavior of the previous section as a task:

```
wallfollowT d_star r =
         liftB (wallfollow1 d_star r)
```

In order to initiate this task, we need to specify a robot structure. As it turns out, tasks implicitly carry this structure[2] along with the computation. The empty task `getRobot` returns it. Thus, we could write a new operator `liftBr` as follows:

```
liftBr ::
   (Robot -> Behavior b) -> Task b e

liftBr a = do {r <- getRobot;
               liftB (a r);
              }
```

Note that if `getRobot` were a function, then we could simple write `liftB (a getRobot);` as it is a task we cannot. This is an example of the language enforcing the fact that tasks can only be combined according to a fixed set of rules.

As `liftBr` expects a behavior parameterized by a robot, we could now write

```
wallfollowTr d_star =
     liftBr (wallfollow1 d_star)
```

to lift `wallfollow1` to a task.

3.2. Task Transformations

To this point, we still cannot generate a task that has both a non-trivial behavior and a terminating event. Task transformations are a means for accomplishing this. We will illustrate the following four transformations in this section:

```
withError     ::
  Event RoboErr -> Task a b -> Task a b
timeLimit     ::
  Time -> Task a b -> Task a (Maybe b)
withB_        ::
  Behavior a -> Task b c -> Task b a
withExit      ::
  Event a -> Task b c -> Task b a
```

The `withExit` transformation provides the means to add a termination event to a task, e.g.

```
wallfollowT1 d_star r =
  withExit
    (predicate (frontsonar r < d_star))
    wallfollowT d_star r
```

The task `wallfollowT1` now performs wall following, but terminates whenever the front sonar detects an obstacle within `d_star` units of distance. We could include an error termination using `withError`:

```
wallfollowT2 d_star r =
  withError
    (predicate (rightsonar r > 2*d_star))
    wallfollowT1
```

This would cause the function to signal an error if the wall were to suddenly recede more than `2*d_star` units from the robot. Likewise, the

[2]Here, we omit details of state manipulation within tasks, but refer the interested reader to [9].

`timeLimit` function aborts a (possibly complex) task if it does not complete within a specified time. It is implemented using `addError` to attach an event to the associated task which occurs at the specified time.

The preceding operators modified the event structure of a task. The `withB_` function is a way of enriching the behavioral component: it defines a behavior to run *in parallel* with a task. When the task exits, the value of the behavior at the time of termination is returned instead of the value of the terminating event. This effectively lets us "piggyback" functionality onto an existing task.

We can illustrate these concepts by extending the previous wall following algorithms to implement the BUG navigation algorithm [10]. Informally, the idea of the BUG algorithm is as follows. When in freespace, the robot drives toward a goal. When an obstacle is encountered, the robot circles the obstacle, looking for the point closest to the goal and the returns to this point to resume travel.

The following code skeleton implements BUG using the wallfollowing task plus two other components:

```
driveTo    ::
    Point2 -> Task WheelControl Bool
atPlace    ::
    Robot -> Point2 -> Event ()
```

The task `driveTo` drives straight toward a goal and returns a boolean event: true when the goal is reached, false when the robot is blocked. The event `atPlace` occurs when the robot is "at" (or at least very close) to a given place.

The top level structure of `bug` relies primarily on sequencing using do:

```
bug goal      =
     do {
         finished <- driveTo goal;
         if (not finished)
             then do {
                      goAround goal;
                      bug goal
                      }
              else nullTask ()
         }

goAround goal =
    do {
       closestPoint <- circleOnceP goal;
       circleTo closestPoint
       }
```

Note in particular that the `if` statement requires both a then and else clause, so `nullTask` is used for the case when the algorithm finishes. Also note the tail-recursive call to bug within the `if`.

We now have to circle the object to find the closest point. Circling can be accomplished as follows:

```
circleTo p =
  withExit (atPlace p) (wallfollowT2 20)

circleOnce =
   do {
       initp <- robotPlace;
       timeLimit 5 (wallfollowT2 20);
       circleTo initp
       }
```

Here, we have "customized" the wall following algorithm in two ways. In `circleTo`, we force termination of the (infinite) `wallfollowT2` behavior when it arrives at place `p`. Now, in `circleOnce`, the `wallfollowT2` task is given a 5 second time limit; just enough time to move away from its initial position. Then, wall following is continued until it returns to its starting point.

One interesting facet of the algorithm is how we determine the closest point to the goal. We use the `atMin` operator:

```
atMin ::
  Behavior a -> Behavior b -> Behavior a
```

This behavior generates the value of its first argument at the time when the second argument is at a minimum. We can now write:

```
closestP r goal =
  atMin (place r)
        (distance (place r) goal)

circleOnceP goal =
    do {
        r <- getRobot;
        withB_ (closestP r goal)
               circleOnce;
       }
```

Here, we use `withB_` to get a "snapshot" of the `atMin` behavior when `circleOnce` terminates. Since it constantly records the robot position at the minimum distance to the goal, the terminating event of `circleOnceP` will contain that value.

In the complete code, note first the pervasive use of `do` to sequence tasks. Likewise, task transformations of a small set of primitives suffice to achieve a fairly rich task behavior. We could go further — for example, note that the `followWall` task has an error condition. We can "catch" this error using `taskCatch`:

```
  taskCatch     ::
     Task a b ->
```

```
    (RoboErr -> Task a b) ->
    Task a b
```

as follows:

```
bugE g = taskCatch (\_ -> bugE g)
                   (bug g)
```

In this case, the bug algorithm now restarts itself on error. Likewise, we could add a time limit to the entire algorithm and include diagnostic code to determine whether the system was somehow "stuck" or otherwise malfunctioning.

3.3. Parallel Tasks

While we have demonstrated sequential task combination, we must also be able to combine tasks in parallel. Imagine, for example, a camera and pan-tilt head on a mobile robot. The overall controller must generate controls for both the camera and the drive wheels. It is often the case that the subsystems for navigation and vision are largely independent: that is, the decomposition of navigation task into subtasks is probably different from the decomposition of the vision controller task. Thus we need an operation that combines two different tasks (each consisting of a different series of atomic tasks) into a single task. Frob includes a number of operators for parallel composition; see [9] for more details. Here, we present one of the simpler parallel composition operators:

```
(|||) :: Task b1 e1 -> Task b2 e2 ->
         Task (b1, b2) (Either e1 e2)
```

This initiates two tasks and defines a continuous behavior that couples the ongoing behaviors of both tasks into a tuple. When either task completes, the composite task also completes, returning a value that identifies the subtask causing termination. An error in either task also terminates the overall task. The `Either` type forms the tagged union of two types.

To illustrate this, suppose we have a task `cameraSweepT` that sweeps the camera at a fixed rate. This task completes when an object of interest, as described by the `Object` type, is detected by the camera. The overall robot controller now needs both a wheel control and a camera control to drive the system, so we must define a task over the type `(WheelControl, CamControl)`. This controller combines the Bug algorithm with a camera sweeping task:

```
cameraSweepT :: Task CamControl Object

goAndWatch ::
  Point2 -> Task (WheelControl, CamControl)
            (Either () Object)
goAndWatch g = bug g ||| cameraSweepT
```

The set of subtasks associated with the two controllers are separate: the controller for the camera is unaware of the subtasks used by the Bug algorithm. The overall task returns either `Left ()` (the `Left` tags the value as the left hand type in the `Either`) or `Right` *obj*, where *obj* is a value of type `Object`. This composite task may be further sequenced with other top-level controls for the system.

It is worth noting that it is easily possible for the tasks to "eavesdrop" on each other using yet another task transformation: `withMyResult`. This transformation makes the behavioral output of a task available for inspection within the task itself.

4. Capturing Architectural Styles

As suggested in the introduction, continuous and discrete behaviors can be thought of as development tools for the lowest level of robotic system software. Tasks provide a level of abstraction for defining more complex algorithms. These notions can be further specialized into abstractions which correspond to specific *architectural styles* such as the subsumption architecture [11], motor schemas [8], port-based agents [1], or the previously cited process model of Lyons [7]. Frob, in and of itself, makes no commitment to a specific style of programming or system architecture. In fact, it is usually straightforward define an architecture or style of programming *within Frob*.

For example, three fundamental notions in the subsumption architecture [11] are "wiring together" modules, suppression, and inhibition. Putting together modules is easily expressed in terms of function composition. Suppression is essentially the notion of allowing one task to take precedence over another under certain conditions. We can easily define a function `suppressedBy x y b` which combines two continuous behaviors `x` and `y` by multiplexing them based on the boolean behavior `b`. So, for example, a function wallfollow1 that at times must be interrupted to support some other task, such as obstacle avoidance, might be expressed simply as:

```
follow3 d r =
  suppressedBy (wallfollow1 d r)
               (avoid r)
               (blocked r)
```

More complex suppression strategies (as well as inhibition) are also easily constructed in Frob (see [12] for details).

As another example, consider Lyons' approach of capturing robotic action plans as networks of concurrent processes [6, 7]. Frob processes can easily mimic Lyons'— for example, here are three of his six compo-

sition operators:

Operator	Lyons	Frob
Sequential	$P; Q$	`do{P; Q}`
Conditional	$P < v >: Q_v$	`do{v < -P; Q v}`
Disabling	$P \# Q$	`P|||Q`

The final primitive, concurrent composition, can be constructed using `|||` in combination with `withMyResult`.

The remaining two operators are defined recursively in his framework by the equations:

$$P :; Q = P : (Q; (P :; Q))$$
$$P :: Q = P : (Q | (P :: Q))$$

They would be similarly defined in Frob.

Another strong point of Frob's declarative approach to robotics programming is the ease in reasoning about Frob programs. We conjecture that Lyons' process algebra, for example, can be proven correct in the Frob framework (thus proving the *implementation* to be correct), but this remains a future research task.

5. Experience and Conclusions

Frob and a sister package, FVision [13], execute on Nomadic Technologies SuperScout II mobile robots. These are differential drive robots that carry three types of sensors: a belt of 16 sonars, bumpers for collision detection, and a video camera. Frob and FVision operate directly on the robots themselves, and Frob operates on the simulator supplied by Nomadic Technologies Inc.

Frob and FVision have been used for our own research development, and have also been used as an instructional tool for an advanced undergraduate course in robotics. The general reaction to this style of programming has been quite positive. Both research personnel and students have found it relatively straightforward to construct systems of moderate complexity using the tools described in this paper. In particular, undergraduates in computer science can easily develop, test and execute algorithms for wall following, mapping, and exploring space within a single semester long course.

From a research perspective, we have found the clarity and code-reduction brought about by Frob to be a significant advantage over both C and C++ or Java [13]. In particular, the resulting programs look very similar to the *specifications* used by engineers and scientists developing algorithms. Furthermore, Frob has the advantage that we are not constrained to a single architectural style. Several may be defined, and even combined within the same application. At the same time, systems defined by Frob are amenable to formal reasoning and program transformation.

We are currently working to broaden our experience with FRP in the robotics domain in a number of ways. First, as noted above we are developing FVision, based on the existing XVision [14] system, to support vision research and integrated vision and robotics. We also plan to develop an environment for robotic hand-eye coordination to test our ideas in a more challenging run-time environment. Finally, we are using Frob to control a team of robots in the Robocup soccer competition.

Acknowledgments

The essentials of FRP were developed by Conal Elliott in conjunction with Fran, and much of the implementation of Frob is based on his work. Thanks also to Gary Ling for developing earlier versions of Frob. This research was supported by NSF Experimental Software Systems grant CCR-9706747.

References

[1] D. Stewart and P. Khosla, "The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects," *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 249–277, 1996.

[2] J. H. J. Salido, J.M. Dolan and P. Khosla, "A modified reactive control framework for cooperative mobile robots," in *Proceedings on SPIE International Symposium on Sensor Fusion and Decentralized Control in Autonomous Robotic Systems*, vol. 3209, pp. 90–100, 1997.

[3] P. Hudak, "Modular domain specific languages and tools," in *Proceedings of Fifth International Conference on Software Reuse*, pp. 134–142, IEEE Computer Society, June 1998.

[4] P. Hudak, S. Peyton Jones, and P. Wadler (editors), "Report on the Programming Language Haskell, A Nonstrict Purely Functional Language (Version 1.2)," *ACM SIGPLAN Notices*, vol. 27, May 1992.

[5] C. Elliott and P. Hudak, "Functional reactive animation," in *International Conference on Functional Programming*, pp. 163–173, June 1997.

[6] D. Lyons and M. Arbib, "A formal model of computation for sensor-based robotics," *IEEE Trans. on Robotics and Automation*, vol. 6, no. 3, pp. 280–293, 1989.

[7] D. M. Lyons, "Representing and analyzing action plans as networks of concurrent processes," *IEEE Transac-*

tions on Robotics and Automation, vol. 9, pp. 241–256, June 1993.

[8] R. Arkin, *Intelligent Robots and Autonomous Agents*. MIT Press, 1998.

[9] J. Peterson and G. Hager, "Monadic robotics," Oct. 1999. To appear in the Proceedings of the Workshop on Domain Specific Languages, Usenix.

[10] V. Lumelsky and A. Stepanov, "Dynamic path planning for a mobile automaton with limited information on the environment," *IEEE Trans. on Automatic Control*, vol. 31, no. 11, pp. 1058–63, 1986.

[11] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Trans. on Robotics and Automation*, vol. 2, pp. 24–30, March 1986.

[12] J. Peterson, "The frob home page." http://www.haskell.org/frob.

[13] A. Reid, J. Peterson, G. Hager, and P. Hudak, "Prototyping real-time vision systems: An experiment in DSL design," in *Proc. 21st Int. Conference Software Engineering*, pp. 484–493, May 1999.

[14] G. D. Hager and K. Toyama, "The "XVision" system: A general purpose substrate for real-time vision applications," *Comp. Vision, Image Understanding.*, vol. 69, pp. 23–27, Jan. 1998.