A Language for Declarative Robotic Programming

John Peterson, Gregory D. Hager, and Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

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. Frob supports a programming style that cleanly separates the **what** from the **how** of a robotic control program. That is, the *what* is a simple, easily understood definition of the control strategy using groups of equations and primitives which combine sets of these control system equations into a complex system. The *how* aspect of the program addresses the unpleasant details, such as the method used to realize these equations, the connection between the control equations and the sensors and effectors in the robot, and communication with other elements of the system. Frob is a system that supports rapid prototyping of new control strategies, enables software reuse through composition, and defines a system in a way that can be formally reasoned about and transformed.

1 Introduction

Over that last several years, the rapid advances in computing and related mechatronics technology have led to a situation where extremely capable (from the point of view of sensors, actuators, and raw computational power) robotic systems are now cheap and widely available. As a result, developing flexible, robust robotic systems is rapidly being reduced to a problem of effective software engineering. In particular, the (as yet) experimental nature of building robotic systems places a strong emphasis on *rapid prototyping* and *software resusability* [17, 12]

It is well-known that the defining quality of any software development environment is its ability to capture the abstractions needed in its domain. Following [6], we can think of robotic software systems consisting of three general categories of algorithms: low-level control (sometimes called the *reactive level* [2]) where continuous, sensor-based feedback loops operate; mid-level programming where low-level actions are combined and sequenced, and high-level task architectures where planning, task monitoring, and user-interfaces operate. In general, different types (and qualities) of tools are available to address software development at each of these levels. For example, low-level behaviors (usually specified using differential or difference equations or dataflow diagrams) are easily implemented using tools like Simulink^{TM} and the RealTimeWorkshop^{TM}. Likewise, robot programming environments such as Colbert [13] or RCCL [9], embedded within imperative languages like C or C++, are generally targeted at mid-level programming. Finally, a variety of architectures have been proposed to handle task-level integration of software modules [4]. In short, most robot programming systems tend to capture one "domain of expertise" well at the expense of the other two.

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. Our approach is based on DSEL (Domain-Specific Embedded Language) technology [11]. The idea of a DSEL is to use a general-purpose language which is customized to deal with a specific domain of interest. Previously, we have used the DSEL approach to construct languages specific to domains such as animation[5], computer music, and computer vision[1].

Frob is a general and flexible framework that supports the following design goals:

Abstraction Frob defines a small set of useful abstractions including events, continuous and discrete behaviors, operational modes, and communication regimes which span the spectrum from low-level control to high-level task specification. We argue that, using these abstractions, Frob makes it easy to think in terms of specifying *what* a program does, not *how* it does it.

Reliability We address reliability on two levels: first, we use an advanced polymorphic type system to ensure that system components are assembled correctly. Type mismatches are detected at compile time instead of during execution. Secondly, we have developed abstractions that automate error handling and recovery schemes.

Modularity Frob allows a complex robotic software system to be cleanly divided into modules that interact over well-defined interfaces.

Rapid Prototyping Frob programs are generally quite compact and easily written and understood, allowing controllers to be rapidly assembled and tested.

It is important to realize that, despite the focus on the abstractions we have built within Frob, it is based on a fully-featured functional programming language (the Haskell language [10]). As a result, Frob is best thought of, not a system architecture, but as a *flexible framework for expressing robot programs* at all levels of abstraction.

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 as Functional Reactive Programming (FRP), a set of primitive operations that define the basic interplay among values expressed as behaviors and events. In this section, we show how behaviors allow for straightforward equational specification of robot behavior.

2.1 Continuous Behaviors

Behaviors are functions over continuous time; they may be computed (sampled) at any time to produce values. For example, the current position of a mobile robot in the plane may be represented as function from continuous time to points in $\Re^2$. The expression of behaviors in Frob is as a polymorphic data type `Behavior` γ where γ is a "placeholder" for the output type of the function. For example, the type `Behavior Point2` denotes behavior of two dimensional points. In Frob, 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. Thus differential equations can be directly expressed (and executed) in a natural manner.

To illustrate how behaviors can be used, we first describe a feedback control algorithm[1] for wall following from 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

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

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.

In rendering this in Frob, we will define a function `wallFollow` which takes the various parameters in the equations above and returns v and ω as a pair of results. Because of the high-level, declarative nature of Frob, this function definition is essentially a rewriting of the above equations in a different syntax, viz:[2]

```
wallFollow vel side front setpoint =
 (v, omega)
   where
    v          = limit vmax (front - setpoint)
    omega      = rerror - derivative side
    rerror     = limit (vel * sin thetamax)
                       (setpoint - side)

limit high x = max (-high) (min x high)
thetamax = 10
vmax     = 100
```

[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.

[2] Frob is a strongly type language, and Frob programs are normally rich with type signatures, but for brevity we omit them in this paper except for functions for which the full definition is not given.

Even to someone who has never seen Frob, the correspondence between this program and the equations above should be very clear.

[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.]

In the above program, `vel` is the (time-varying) value of the current robot velocity, and `front` and `side` are two (time-varying) sensor behaviors (the front and side sonar, respectively). The parameter `setpoint` is the minimum distance to an obstacle in front of the robot as well as the nominal desired distance to the wall. 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 run this program we need to define a mapping from robot sensors to robot effectors. As our code currently runs on a Nomadic Tech SuperScout II (see Section 5), Frob defines the following functions:

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

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 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.

Thus to "package" the wall follower with the appropriate sonar for following a wall on the right we write:

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

follower d r =
   wallFollow (velocity r)
     (frontsonar r) (rightsonar r) d
```

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

```
runScout (follower 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. This highlights the modularity afforded by Frob programming.

2.2 Discrete Behaviors

In practice, low-level control algorithms are often designed and almost always executed as discrete-time systems. Furthermore, many recursive estimation algorithms, *e.g.* the Kalman filter [7], are specified using a set of recursive discrete-time equations.

In order to permit direct rendering of such equations, Frob supports *discrete behaviors*. As with continuous behaviors, discrete behaviors model values changing over time, but now the changes are at a fixed sampling rate controlled by a clock. Discrete behaviors may be combined in the same manner as continuous behaviors: functions such as `+` and `*` are available in the discrete world as well as the continuous one. However, discrete behaviors carry an extra constraint: behaviors combined by operators such as `+` must share a common clock (*i.e.* have the same sampling rate and phase). In the type language of Frob, these behaviors are denoted by `DBehavior` *clock type*, where *clock* is a type defining the clock associated with the behavior and *type* is the value of the behavior. In this way, the underlying type system can catch, at compile time, programming errors in which unsynchronized data sources are combined.

Discrete behaviors support two operations not applicable to continuous behaviors: `delay` and `interact`. The `delay` function creates a discrete behavior delayed by one clock interval. It is defined as

$$\begin{aligned} (\texttt{delay}\ x_0\ y)_0 &= x_0 \\ (\texttt{delay}\ x_0\ y)_k &= y_{k-1} \quad k > 0 \end{aligned}$$

where β_k is the value of discrete behavior β at time k. An important property of `delay` is that it preserves the clock: that is, the result of `delay` is based on the same clock stream as its input.

The function `interact` is a means of performing an interaction with the outside world at every clock tick. It is defined as

$$(\texttt{interact}\ f\ b)_k \quad = \quad f(b_k)$$

This resembles the "usual" lifting of a scalar operation into the behavioral domain via sampling except for the fact that f is allowed to perform I/O. The clock implicit in discrete behaviors (but not continuous ones) defines precisely the times at which f will interact with the outside world. Most functions in Frob are *pure*: that is, they interact only through their parameters and results, not via side-effects. This very important property supports compositionality: it prevents unintended interactions between components. The `interact` function allows communication with the outside world in a controlled manner that does not destroy compositional properties.

Behavioral functions that do not use either `delay` or `interact` may be used for both discrete and continuous behaviors. Indeed, the `wallFollow` function could be used with discrete behaviors rather than continuous ones with absolutely no change.

We illustrate the use of discrete behaviors to define a simple visual tracking algorithm based on calculating the centroid of a bright object. We first define a function which accepts a sequence of images (a discrete behavior) and produces a sequence of displacements, `delta`, indicating the location of the bright spot relative to the center of the image:

```
centroid im thresh = center - (sizeof im) / 2
 where
  binIm = threshIm im thresh
  center = calc_centroid binIm
```

To turn this into a visual tracking algorithm, we simply accumulate these changes over time:

```
blobTrack source initPos thresh = pos
 where
  image = interact (acquire source size) pos
  delta = centroid image thresh
  pos   = delay initPos (pos + delta)
```

Note that this function encodes an explicit recursive dependency — `pos` is defined in terms of `delta` and `pos` delayed one time step. The laziness of the underlying functional language allows us to write such recursive specifications. The function `interact` is used to acquire the portion of the current video image at the position `pos` at each iteration of the discrete behavior.

The result of this function is a (discretely varying) position for a blob which is initially at `initPos`. The clock rate is 30Hz—the frame rate of the underlying camera system. This clock is independent (indeed it operates at a different rate) than that of the underlying robot control system. Thus, if we wish to control the robot using camera information, we need to resample the signal. This is handily accomplished by the `smooth` function which converts a discrete behavior into a continuous one. Using `smooth`, `blobTrack` and a function `pointx` which returns the first (the "x") coordinates of a 2-D point, we can write and execute a system for "staring" at a given blob as follows:

```
startpt = (100,100)
thresh  = 200

stare pt r = (0,smooth val)
  where
    loc = blobTrack (video r) pt thresh
    val = - pointx (loc - size (video r)/2)

runScout (stare startpt)
```

Here, the resampling of the (now continuous) control signal produced by `stare` is hidden "inside" `runScout` (which, of course, expects a continuous behavior). It is important to note that there is nothing fundamental about this choice — we could have just as easily defined `runScout` on discrete behaviors synchronized to the underlying control system clock.

3 Sequencing and Processes

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 unclocked events rather than behaviors. This motivates Frob's notion of an *event*. An event stream is a sequence of values each occurring at a specific time. Event streams are somewhat similar to behaviors but they are undefined at times other than the event occurances.

Using events, we show how to define one example of a higher-level programming construct: the *process* [14, 15]. A process defines a behavior that either terminates, returning a value on termination, or aborts. Thus, using processes, we are able to move from completely reactive algorithms to more task-driven ones that operate in a sequential manner.

3.1 Events and Reactions

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 occurances 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 backquotes around `untilB` designate it as an infix operator. The `-=>` operator couples an event with response

The next example uses reactivity to augment the wall following behavior defined earlier. In this example, we pass control to a new behavior when the robot encounters either an interior or exterior corner.

```
follow1  intB extB setPoint r =
 follow setPoint r `untilB`
   predicate (rightsonar r  > sideThresh)
     -=> extB
            .|.
   predicate (frontsonar r < frontThresh)
     -=> intB
```

This can be read: "Follow the right wall until either the side reading is greater than `sideThresh` (and then execute `extB`) or until the front reading is less than `frontThresh` (and then follow behavior `intB`). Note that we reuse the control system defined earlier, `follow`, by adding reactions that terminate the wall following behavior and pass control to the next process at hand. In the following section, we generalize this pattern using *processes*.

3.2 Processes

A common abstraction for programming above the level of primitive behaviors is to define a (possibly infinite) "chain" of actions using, for example, a deterministic finite automaton or similar graphical abstraction [2]. Frob uses the type `Process cv ev` to denote processes that define a continuous value of type `cv`, and finish by delivering an end value of type `ev`. Processes are created using `makeProcess` which accepts three arguments: a behavior (either continuous or discrete), a completion event, and an abort event. The type `ProcessAbort` carries information from the aborting event to the handler associated with aborted processes. For example, this function encodes the wall follower as a process:

```
sonarterm tooclose toofar r =
   (predicate (rightsonar r  > toofar)
     -=> True
            .|.
   predicate (frontsonar r < tooclose)
     -=> False)

follow2 setPoint =
  makeProcess
   (follow setPoint)
   (sonarterm (setPoint / 2) (2 * setPoint))
   neverE  -- this process never aborts
```

This definition is nearly identical to `follow1`; we have merely removed the `untilB` and the continuation argument. Sequencing is no longer explicit in the definition of the process.

Processes are chained together using a *sequencer* [3]. Frob uses notation `do {m1;m2}` to denote sequential composition of processes `m1` and `m2`. This mirrors the type of imperative sequencing common in traditional programming languages. The following example demonstrates the chaining of processes:

```
patrol setPoint =
  do {e <- follow2 setPoint;
      if e then turn -90  -- right
           else turn  90; -- left
      patrol setPoint}
```

where `turn n` turns the robot through `n` degrees. The value returned on process completion is bound by `<-`. Thus, when executed, this program follows a wall through inside and outside corners, turning the robot

[3] In this case, we use a style of sequencer called a *monad* which is well-known in the functional programming community [16].

90 degrees every time it is blocked or looses the wall. Note the use of the value returned from `follow2` to select the next process and the tail-recursive call of patrol to continue process execution.

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 robotics systems software. Events and processes support development at the second level. At the highest level, we are interested in developing abstractions which correspond to specific *architectural styles* such as the subsumption architecture [3], motor schemas [2], port-based agents [17], or the previously cited process model of Lyons [15].

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*. The advantages of using Frob to define this level of abstraction are two-fold:

1. We are not constrained to a single architectural style. Several may be defined, and even combined within the same application. And of course we are free to invent new ones.
2. Although many architectural styles have been proposed, few have been implemented as new languages; rather, they are encoded in C, C++ or Lisp with the architectural style being used as a guideline for structuring the code. The same mechanisms that permit concise and elegant expression of lower level functionality in Frob permit the expression of these higher-level abstractions as a domain specific embedded language.

For example, three fundamental notions in the subsumption architecture 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. For example, 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 wall follower that at times must be interrupted to support some other task, such as obstacle avoidance, might be expressed simply as:

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

More complex suppression strategies (as well as inhibition) are also easily constructed in Frob.

As another example, consider Lyons' approach of capturing robotic action plans as networks of concurrent processes [14, 15]. Frob processes are can easily mimic Lyons'— for example, here are four of his six composition operators:

Operator	Lyons	Frob
Sequential	$P;Q$	$\texttt{do}\{\texttt{P};\texttt{Q}\}$
Conditional	$P < v >: Q_v$	$\texttt{do}\text{–}\texttt{v} < -\texttt{P};\texttt{Q}''$
Concurrent	$P \mid Q$	$\texttt{P} \mid \texttt{Q}$
Disabling	$P\#Q$	$\texttt{P\#Q}$

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

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

The lazy evaluation semantics of Frob permits us to write these same equations directly, only with different syntax:

```
P <> Q = do { v<-P; Q; (P<>Q) }
P >< Q = do { v<-P; (Q | (P><Q)) }
```

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 Implementation and Experience

We have developed Frob and a sister package, FVision [1], to execute on Nomadic Technologies SuperScout II's. These robots carry three types of sensors: a belt of 16 sonars, bumpers for collision detection, and a video camera. The drive mechanism uses two independent drive wheels and is controlled by setting the forward velocity and turn rate. The robot also provides a continuous estimate of position using dead-reckoning. Here, we describe how Frob is implemented and some of our experiences

5.1 Implementing Frob

We have implemented Frob on top of basic motion control and sensor libraries supplied by Nomadic

Technologies. Frob itself is implemented as a Haskell library. We execute Frob programs using the Hugs system, a small, portable Haskell interpreter. Low level robot control tasks are carried out by C++ code imported directly into the Haskell interpreter. In addition, the running Frob program interacts with a remote console window via telnet, allowing the program to respond to keyboard input as well as its sensors. Robots may also send messages to each other via the radio modem, allowing us to explore cooperative behaviors.

To date, we have built a number of demonstrations using Frob and FVision. Briefly, our experiences to date are:

- Overall performance has not been a problem, in spite of the fact that we are running Haskell using an interpreter (Hugs) with garbage collection. This is, in part, because we are generally I/O bound: the low-level robot libraries have a limited communication bandwidth (about 15Hz) with the motor control and sensing hardware. In the case of FVision, it is because most of the computation takes place in imported image-processing libraries (written in C++) and not in Hugs.

- Frob programs are far smaller than C++ programs for the (relatively small) tasks developed so far. More importantly, most of the programs are far easier to understand "at a glance" than the corresponding C or C++ programs.

- The current system assumes that computation can consume events as soon as they arrive. Thus, bumper strikes, for example, are handled as soon as they are sensed. If the system becomes swamped for some reason, there is currently no support for executing a "context switch" which would provide for immediate reaction to high-priority events. This is currently the only major limiting factor we have discovered in the system.

One of the most positive aspects of our experience with Frob is that it provides a way of rapidly prototyping and testing a wide range of control algorithms. In this way, Frob provides an extremely useful design and analysis tool for experimental robotics.

5.2 From Prototype to Production

One potential objection to Frob is that, while it is ideal for exploring a design space for systems, it is not (at least in its current state) well-suited for a real-time setting where, for example, garbage collection is difficult to support and runtime support is minimal. Furthermore, history would suggest that the bulk of low-level robot programming will continue to be performed in traditional languages such as C and C++, not in Haskell.

To counter this, we point out that, once we have developed a system in Frob, portions of the system can often be reflected with minor change into the C++. As a particular example, here is the code for the function `blobTrack` defined in Section 2 in a C++ environment which supports discrete behaviors:

```
#define THRESH 200
extern Pos2 centroid(const Image &,int);
main()
{
   // Video source
  Meteor v();

  // ``lift'' an existing function to pipes
  DBLift2<Pos2,Image,int> centroidDB(centroid);

  DBehavior pos<Pos2>, image<Image>, delta<Pos2>;

  image = acquireDB(v,100,100,pos);
  delta = centroidDB(image,THRESH);
  pos   = delayDB(initPos,pos + delta);

  runpipe(display(pos));
}
```

The result of this program is a stream of blob coordinates starting at $(100, 100)$. Note that the only substantial differences between this program and the Frob version are: 1) we have had to explicitly type every variable (C++ is not polymorphic), and 2) we have included an explicit "lifting" of an existing function (`centroid`) to operate on streams of data. In fact, we also perform this lifting in Frob; it is simply hidden in the examples presented above by overloading.

6 Summary and Conclusions

We would argue that Frob has succeeded in a number of ways:

- In our algorithms, the low-level details implementing time-based reactivity has been largely hidden. The resulting programs look very similar to the *specifications* used by engineers and scientists developing algorithms.

- We have been able to define a number of programming abstractions which not only support natural specification of robot behavior, but also tend to make the code compact, clear and highly reusable.

- The pervasive use of Functional Reactive Programming at all levels of the system allows other systems based on FRP such as the FVision system to be integrated into this framework.

- 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 [8] 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.

Due to space limitations, we have been forced to omit some details of the coded examples. We invite the reader to consult `http://haskell.org/frob` for full details of our code.

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] G. Hager A. Reid, J. Peterson and P. Hudak. Prototyping real-time vision systems: An experiment in dsl design. DCS RR-1164, Yale University, New Haven, CT, October 1998.

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

[3] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Trans. on Robotics and Automation*, 2(1):24–30, March 1986.

[4] E. Coste-Maniére and B. Espiau, editors. *International Journal of Robotic Research, Special Issue on Integrated Architectures for Robot Control and Programming*, volume 17, 1998.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[6] J. Faymann, E. Rivlin, and H. I. Christensen. A system for active vision driven robotics. In *Proc. 1996 Conf. on Robotics and Automation*, pages 1986–1992, April 1996.

[7] Arthur Gelb, editor. *Applied Optimal Estimation*. MIT Press, Cambridge, MA, 1974.

[8] G. D. Hager and K. Toyama. The "XVision" system: A general purpose substrate for real-time vision applications. *Comp. Vision, Image Understanding.*, 69(1):23–27, January 1998.

[9] V. Hayward and J. Lloyd. *RCCL User's Guide*. McGill University, Montreál, Québec, Canada, 1984.

[10] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[11] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[12] J. Hampshire 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*, volume 3209, pages 90–100, 1997.

[13] K. Konolige. Colbert: A language for reactive control in sapphira. In G. Brewka, C. Habel, and B. Nebel, editors, *Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*. Springer, 1997.

[14] D. Lyons and M. Arbib. A formal model of computation for sensor-based robotics. *IEEE Trans. on Robotics and Automation*, 6(3):280–293, 1989.

[15] Damian M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(7):241–256, 1993.

[16] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages*. ACM, January 1993.

[17] D.B. Stewart and P.K. Khosla. The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277, 1996.