System Presentation – Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages

Izzet Pembeci
Johns Hopkins University
Baltimore, MD
pembeci@cs.jhu.edu

Henrik Nilsson
Yale University
New Haven, CT
nilsson@cs.yale.edu

Gregory Hager
Johns Hopkins University
Baltimore, MD
hager@cs.jhu.edu

ABSTRACT

Software for (semi-) autonomous robots tends to be a complex combination of components from many different application domains such as control theory, vision, and artificial intelligence. Components are often developed using their own domain-specific tools and abstractions. System integration can thus be a significant challenge, in particular when the application calls for a dynamic, adaptable system structure in which rigid boundaries between the subsystems are a performance impediment. We believe that, by identifying suitably abstract notions common to the different domains in question, it is possible to create a broader framework for software integration and to recast existing domain-specific frameworks in these terms. This approach simplifies integration and leads to improved reliability. In this paper, we show how Functional Reactive Programming (FRP) can serve as such a unifying framework for programming vision-guided, semi-autonomous robots and illustrate the benefits this approach entails. The key abstractions in FRP, reactive components describing continuous or discrete behavior in a declarative style, are first class entities, allowing the resulting systems to exhibit a dynamic, adaptable structure which we regard as especially important in the area of autonomous robots.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Languages, Design

Keywords

Functional Programming, Robotics, Vision, Domain-Specific Languages

PPDP'02, October 6-8, 2002, Pittsburgh, Pennsylvania, USA.

1. INTRODUCTION

Developing practical software for autonomous or semi-autonomous robots is a complex task. It involves the integration of concepts spanning multiple disciplines, including control theory, hybrid systems, vision, artificial intelligence, and human-computer interaction. Within each of these areas, languages or support libraries that capture domain-specific functionality, design patterns, and abstractions are used to structure the solution and aid in the reuse of actual code. For example, the language Simulink is very popular among control engineers, in part since it allows working control system code to be derived more or less automatically from the control equations describing the controller. Another example is XVision2 [6, 13], a sophisticated C++ library containing components for building advanced vision algorithms, for example for identification and tracking of objects in video streams.

As an application is developed, these software components must be integrated into a seamless whole. Given the diversity of the domains, this is less than straightforward: the employed domain-specific support tools often (and arguably rightfully so) pay little attention to concerns not immediately relevant to the domain in question. A common integration paradigm is to treat each subsystem as a black box that only communicates through a simple procedural interface. The overall system is then composed from such black boxes through suitable "glue code", expressed in some "least-common denominator" language like C, or a scripting language like Perl or Python. The problems with this approach are twofold. First, composition at such a low abstraction level generally cannot take advantage of the relatively high level of abstraction available *within* the black boxes. Thus, for compositional purposes, the value of those abstractions is lost. Second, the resulting system structure tends to be too rigid: there is only so much one can do with black boxes communicating via low-level interfaces.

Our approach is to instead look for common notions of abstraction across the involved domains. By expressing interfaces in these higher-level terms, system integration can be greatly simplified. This is especially true if the notions are declarative, since that generally promotes compositionality. Furthermore, by providing a seamless link between the black boxes and the outer world, we can often expose much more of their inner functionality. In effect, the black boxes are replaced by *tool boxes* that provide for a more flexible, dynamic, and adaptable system structure.

In the area of robotics, reactivity, continuous behavior, discrete behavior, and switching among behaviors are examples of declarative notions which either are common to many of the involved domains, or in terms of which domain-specific constructs naturally can be expressed. *Functional Reactive Programming* (FRP) [21] is a framework for expressing reactive systems which captures exactly these notions. Earlier work has shown how FRP can beneficially be instantiated to address some of the key areas mentioned above in isolation, such as robotics programming at the control-system and task levels [18, 17] and vision [19].

In this paper, we show how these ideas can be scaled to create a fully functional, vision-guided navigation system operating on a real mobile robot equipped with stereo vision. On a practical level, we claim that FRP can incorporate large amounts of external functionality, and combine it in an efficient and reliable manner. On a conceptual level, we aim to demonstrate the ease of system integration and the flexibility of the resulting system. The latter results from the fact that the reactive components describing the system behavior are *first class entities* in the FRP framework, allowing for a very flexible and adaptable system structure. FRP is usually implemented as an embedding inside a host language. In this paper the host language is Haskell, which is the most common choice to date.

We substantiate these claims by describing *Frob* (Functional Robotics), an instantiation of the FRP framework for robot programming, and by describing a tracking-based navigation system written in Frob using the XVision2 library [6, 13]. Frob illustrates how some application domains can be handled within the FRP framework by adding domain-specific abstractions. We show that Frob is flexible enough to subsume a number of extant robot programming paradigms, allowing the programmer to pick the right approach for the task at hand, and even to mix and match paradigms thanks to the common underlying FRP framework. The tracking example illustrates multi-domain system integration in the context of working code for our vision-equipped, mobile robot. The XVision2 library is imported and lifted to the FRP level, resulting in *FVision*, a flexible tracking tool box. In addition, we show how a rudimentary human-computer interface can be integrated into the same reactive framework, thus providing for semi-autonomous, vision-guided, human-in-the-loop operation.

The rest of this paper is organized as follows. Section 2 gives an overview of our hardware setup and the software architecture. Section 3 gives an introduction to *Arrowized FRP* (AFRP), the particular version of FRP we are using in this paper, and Frob. Section 4 then shows how Frob can be used for various styles of robot programming. Section 5 describes the tracking-based navigation system. Section 6, finally, sums up and gives conclusions.

2. SYSTEM OVERVIEW

This section gives a brief overview of our robot system and its software architecture. Figure 1 shows the hardware organization. The actual robot system consists of the robot itself, an ActivMedia Robotics Pioneer 2; the on-board computer, an x86 laptop running Linux; and a stereo camera.

The Pioneer 2 is a two-wheeled (differential-drive) mobile robot. It has an on-board microcontroller running a proprietary operating system that is responsible for the low-level operation of the robot. While the microcontroller could be programmed to perform some (not too complex) high-level tasks, we only use it in a command-driven mode. In this case, the microcontroller carries out simple movement commands sent to it from the on-board laptop computer over a serial link. Status information and position information obtained through dead-reckoning are sent in the other direction, from the microcontroller to the computer.

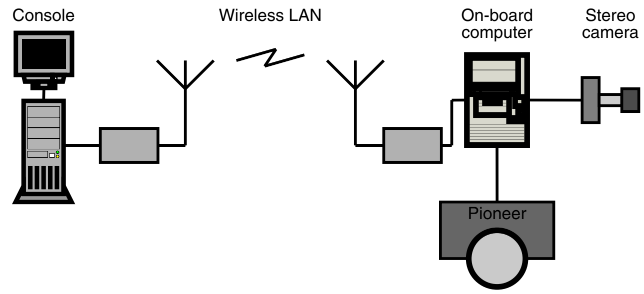

Figure 1: Overview of the hardware setup.

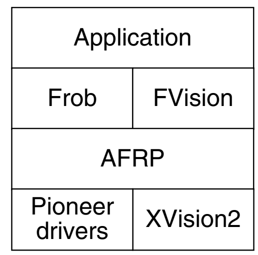

Figure 2: Software architecture.

The on-board computer is thus responsible for the high-level operation of the robot, along with vision processing and communication with the outside world. The stereo camera is connected to the computer via a fast firewire interface. Stereo vision enables us to recover depth information from the video stream which in turn allows the vision system to detect objects through their 3-dimensional shape as well as appearance, e.g. pre-arranged color cues.

Communication with the outside world takes place over a wireless LAN, permitting the robot to move about without any attached cables. The robot usually operates semi-autonomously under the supervision of an operator at a console. The on-board computer feeds live video to the console where the operator can instruct the robot to carry out certain tasks such tracking an object while avoiding obstacles or moving to some particular location. If necessary the operator can take direct control and tele-operate the robot.

Figure 2 shows the software architecture of the on-board system. At the lowest level, there are two subsystems. The Pioneer drivers are a set of routines for communicating with the microcontroller. XVision2 handles the camera interfacing and all computationally demanding vision processing and manages the system console. Interfaces to these two subsystems are lifted into the reactive AFRP level, and two domain-specific, languages are then built on top: Frob for robot programming and FVision for describing vision processing algorithms. Since these two languages are built on a common framework it is possible to create a very tightly coupled interface between the vision and robot control systems.

3. THE SOFTWARE FRAMEWORK

Functional Reactive Programming (FRP) is a declarative framework for describing hybrid (continuous and discrete) systems. In this paper we use the latest full-scale implementation of FRP is called *Arrowized FRP* (AFRP). This section gives a short introduction to AFRP; see [15] for a fuller account. At present, the only implementation of AFRP is embedded in the Haskell programming language; examples in this paper are written in a modified Haskell syntax. AFRP includes a special syntax called *arrow notation* that is not a standard part of Haskell. This notation makes AFRP programs much easier to comprehend and is described below. We have tried not to assume detailed knowledge of Haskell in our examples. Nevertheless, some familiarity with modern functional programming notation might be helpful. Finally, this section describes the basics of Frob (features of FVision are introduced as needed in section 5).

3.1 AFRP

Two key concepts in FRP are *signals* and *signal functions*. Signal functions, which are *first class entities*, operate on one or more input signals, producing one or more output signals. An FRP system consists of a number of interconnected signal functions, operating on the system input, or stimuli, and producing the system output, or response. The signal functions operate in parallel, sensing a common *rate* of time flow. The structure of an FRP system may evolve over time. For example, new signal functions can be added or old ones deleted. These structural changes are known as *mode switches*. The first class status of signal functions in combination with powerful switching constructs make FRP unusually flexible as a language for describing hybrid systems.

A *signal* is, conceptually, a function of time, or, equivalently, a time-varying value (sometimes called *fluent*). The domain of a signal can either be continuous or discrete. In the former case, the signal is defined at every point in time. In the latter case, the signal is a partial function, only defined at discrete points in time. Such a point of definition is called an *event*.

An executable FRP implementation can only approximate this conceptual signal model since continuous-time signals necessarily are evaluated for only a discrete set of sample points (the points need not be equidistant, but it is the *same* set for all continuous-time signals in a system). Thus, once the sample points have been picked, the true value of continuous signals between these points are operationally irrelevant: the sampled signals operationally become the true signals. We could still consider sampled continuous-time signals to have *some* value between sampling points if we like (for example, the value at the closest preceeding sampling point, or some interpolated value), but current FRP implementations are defined in such a way that this consideration is irrelevant.

By sampling sufficiently densely, or by picking sampling points cleverly, one can hope to obtain a "good" approximation. Indeed, by imposing certain limitations on the system, it is possible to guarantee convergence results in the limit (at least under the assumption of exact arithmetic) [21]. While interesting, these aspects are outside the scope of the present paper. We will assume that a system is executed at a high enough rate to ensure appropriate responses, as long as the input signals are limited to a sufficient bandwidth.

As to the implementation of discrete-time signals, each is defined on a *subset* of the sampling points (not necessarily a proper subset, but in general a *distinct* subset for each discrete-time signal). AFRP captures the notion of a partial function by lifting the range using an option type called `Event`. This type has two constructors: `NoEvent`, representing the absence of a value; and (also) `Event`, representing the presence of a value. In Haskell notation:

```
data Event a = NoEvent | Event a
```

We can intuitively think of continuous-time signals as functions of type

$$\texttt{Time} \rightarrow A$$

for some suitable representation type `Time` for continuous time, and some value type A, whereas discrete-time signals can be understood as functions of type

$$\texttt{Time} \rightarrow \texttt{Event}\, A$$

However, signals are *not* first class entities in AFRP; in particular, signals do not possess a type.

A *signal function* is a *pure* function that maps a stimulating signal onto a responding signal while satisfying certain causality constraints (roughly, the output must not depend on future input). Unlike signals, signal functions *are* first class entities in AFRP. This means that they are treated just like other values in the language: they have a type, they can be bound to variables, they can be passed to and returned from functions, etc. The type of a signal function mapping a signal of type A onto a signal of type B is written $\texttt{SF}\, A\, B$. Intuitively, we have

$$\texttt{SF}\, A\, B = (\texttt{Time} \rightarrow A) \rightarrow (\texttt{Time} \rightarrow B)$$

If more than one input or output signal is needed, tuples are used for A and B since a signal of tuples is isomorphic[1] to a tuple of signals.

The strict distinction between signals and signal functions is a defining feature of AFRP. In some earlier versions of FRP, signals could "masquerade" as signal functions, thus effectively giving signals first class status. The upshot of the distinction is that the value of a signal can only be accessed pointwise in AFRP: it is not possible to refer to the entire signal at once. Our experience thus far suggests that AFRP is less susceptible to space and time leaks found in previous FRP implementations. This, in turn, has been a key factor in our ability to scale FRP to real applications.

Arrowized FRP is an instance of the arrow framework proposed by Hughes [8]. A type constructor of arity two together with three operations, `arr`, `>>>`, and `first`, form an arrow provided certain algebraic laws hold. Ordinary functions is the canonical example of an arrow. In AFRP, the type constructor `SF` is an arrow instance. The three arrow operations have the following type signatures for `SF`:

```
arr   :: (a -> b) -> SF a b
(>>>) :: SF a b -> SF b c -> SF a c
first :: SF a b -> SF (a,c) (b,c)
```

The combinator `arr` *lifts* an ordinary function to a signal function by applying the function pointwise to the input

[1] Modulo extra bottom elements due to the non-strict semantics of Haskell.

signal. The result is a *stateless* signal function since the instantaneous value of the output signal at any point in time only depends in the instantaneous input value at that same time. `>>>` is a serial composition operator, similar to ordinary function composition. `first` is a "plumbing" combinator, making it possible to route signals in a network of signal functions.

Other arrow combinators can be defined in terms of these primitives. A commonly used derived combinator is `&&&`:

```
(&&&) :: SF a b -> SF a c -> SF a (b,c)
```

This combinator composes two signal functions in parallel (spatially), feeding the same input signal to each of them and pairing the two resulting output signals.

Another important combinator is `loop`:

```
loop :: SF (a,c) (b,c) -> SF a b
```

The `loop` combinator is used for recursive definitions: the `c`-part of the output is fed back to the input. Note that `loop` cannot be defined in terms of the three basic arrow combinators. The inclusion of `loop` means that AFRP is an instance of the class of "loopable" arrows.

Although signals are not first class values in AFRP, Paterson's syntactic sugar for arrows [16] effectively allows signals to be named. This eliminates a substantial amount of plumbing, resulting in much more legible code. In this syntax, an expression denoting a signal function has the form:

`proc` *pat* `->` `do` [`rec`]
    pat_1 `<-` $sfexp_1$ `-<` exp_1
    pat_2 `<-` $sfexp_2$ `-<` exp_2
    ...
    pat_n `<-` $sfexp_n$ `-<` exp_n
    `returnA -<` *exp*

The keyword `proc` is analoguous to the λ in λ-expressions, *pat* and pat_i are scalar patterns binding signal variables pointwise by matching on instantaneous signal values, *exp* and exp_i are scalar expressions defining instantaneous signal values, and $sfexp_i$ are expressions denoting signal functions. The idea is that the signal being defined pointwise by each exp_i is fed into the corresponding signal function $sfexp_i$, whose output is bound pointwise in pat_i. The overall input to the signal function denoted by the `proc`-expression is bound by *pat*, and its output signal is defined by the expression *exp*. The signal variables bound in the patterns may occur in the scalar expressions, but *not* in the signal function expressions ($sfexp_i$). If the optional keyword `rec` is used, then signal variables may occur in expressions that textually precedes the definition of the variable, allowing recursive definitions (feedback loops). The syntactic sugar is implemented by a preprocessor which expands out the definitions using only the basic arrow combinators `arr`, `>>>`, `first`, and, if `rec` is used, `loop`.

For a concrete example, consider the following:

```
sf = proc (a,b) -> do
    c1 <- sf1 -< a
    c2 <- sf2 -< b
    c  <- sf3 -< (c1,c2)
    d  <- sf4 -< b
    returnA -< (d,c)
```

Here we have bound the resulting signal function to the variable `sf`, allowing it to be referred by name. Note the use of the tuple pattern for splitting `sf`'s input into two "named signals", `a` and `b`. Also note the use of tuple expressions for pairing signals, for example for feeding the pair of signals `c1` and `c2` to the signal function `sf3`.

AFRP provides a rich set of functions for operating pointwise on events. In fact, the type `Event` is abstract in the current AFRP implementation, so events cannot be manipulated except through these operations. The following selection of event operations is used in this paper:

```
tag     :: Event a -> b -> Event b
lMerge  :: Event a -> Event a -> Event a
rMerge  :: Event a -> Event a -> Event a
filterE :: (a -> Bool) -> Event a -> Event a
```

The function `tag` tags an event with a new value, replacing the old one. `lMerge` and `rMerge` allow two discrete signals to be merged pointwise. In case of simultaneous event occurrences, `lMerge` favours the left event (first argument), whereas `rMerge` favours the right event (second argument). `filterE`, finally, suppresses events which do not satisfy the boolen predicate supplied as the first argument.

3.2 Frob

Frob programs are signal functions operating on robot-specific input and output types. These types capture the relevant aspects of available sensors and actuators. Frob has mechanisms for abstracting over classes of available functionality, allowing generic code that is not tied to a specific hardware platform to be written. However, in this paper, we will target our Pioneer platform explicitly to keep things simple. The input and output types for the Pioneer are `PioneerInput` and `PioneerOutput`. Consequently, a top-level signal function for controlling the Pioneer has the type

```
SF PioneerInput PioneerOutput
```

The function `reactimatePioneer` is provided to connect such a signal function to the sensors and actuators of the robot:

```
reactimatePioneer ::
    SF PioneerInput PioneerOutput
    -> IO ()
```

It forms the input signal pointwise by reading the various sensors, applies the signal function to the current input value to get the current output, and then uses that output to send the appropriate commands to the actuators.

The `PioneerInput` and `PioneerOutput` types are essentially record types with one field for each available sensor and actuator. For example, `PioneerInput` has fields for odometry, the images from the stereo camera, and input from the console such as mouse position and keyboard input. `PioneerOutput` has fields for controlling the speed of the robot and for overlaying graphics on the live video stream.

AFRP provides a framework for constructing records incrementally by merging sets of fields, as opposed to constructing a record by providing values for all fields at once. This idea is captured by the type class `MergeableRecord`. Any record type that supports this form of record construction, such as `PioneerOutput`, is made an instance of this class. Functions for constructing individual fields are also provided. These can then be merged into sets of fields, and eventually finalized into a record:

```
mrMerge :: MergeableRecord a =>
    MR a -> MR a -> MR a
mrFinalize :: MR a -> a
```

Some of the functions available for controlling different aspects of the Pioneer platform are listed below. Note that each returns a *mergeable* `PioneerOutput` record:

```
ddBrake :: MR PioneerOutput
ddVelDiff :: Velocity -> Velocity
             -> MR PioneerOutput
ddVelTR :: Velocity -> RotVel
           -> MR PioneerOutput
fvgoOverlaySG :: SimpleGraphic
                 -> MR PioneerOutput
```

The prefix `dd` stands for *Differential Drive*, and `fvgo` stands for *FVision GUI Output*. `ddBrake` brakes the two wheels. `ddVelDiff` sets the desired peripheral wheel velocity (in m/s) for the left and right wheel, whereas `ddVelTR` sets the wheel velocities in terms of the overall desired translational and rotational velocity of the robot. Only one mode of controlling the wheel velocities can be used at any one point in time. If two or more are used, one will override the others. `fvgoOverlaySG`, finally, is used to overlay simple graphics (lines, circles, text, ...) on the live video stream sent to the console. Merging of multiple `fvgoOverlaySG` is by superposition. The semantics of merging is thus field-dependent.

The following is a simple Frob program that instructs the robot to move forward at a constant speed, while superimposing a circle over the live video:

```
fwdDemo :: SF PioneerInput PioneerOutput
fwdDemo = proc inp -> do
    (x,y) <- mousePos -< inp
    returnA -< mrFinalize $
        ddVelTR 0.25 0
        `mrMerge` fvgoOverlaySG (SGEllipse
                                 (x-10,y-10)
                                 (x+10,y+10)
                                 True
                                 Red)

main = reactimatePioneer fwdDemo
```

`mousePos` extracts the mouse position from the input signal. The circle is centered at these co-ordinates, causing it to track the mouse movements.

3.3 Tasks

Nearly all robot programming languages have a notion of *task*: actions to be carried out by the robot to accomplish a specific goal. Tasks thus involve one or more continuous activities, such as driving, and some way of determining when the task has been completed. There are also ways of combining tasks into more complex tasks.

Hager and Peterson defined a task notion for an earlier version of Frob [5]. They defined a task as a combination of a continuous behavior and a terminating event. Such tasks form a monad, allowing Haskell's monadic constructs such as `do` notation to be used for task-based programming.

AFRP provides a similar notion of task, where a task is seen as a signal function operating on a continuous-time input signal and producing a pair of a continuous-time output signal and an event signal for indicating termination. For an input signal of type A, an output signal of type B, and a terminating event of type C, the type of a task is written $\texttt{Task}\,A\,B\,C$, and we intuitively have

$$\texttt{Task}\,A\,B\,C = \texttt{SF}\,A\,(B\texttt{,Event}\,C)$$

Two tasks are sequenced by switching from the first signal function to the second on the first event on the first signal function's event output. Thus switching is what causes termination: a signal function as such never terminates.

`Task` is an instance of `Monad`, and Haskell's `do`-notation for monads can thus be used to express sequencing:

```
task1 = do
    taskexp1
    taskexp2

task2 = do
    x <- taskexp3
    taskexp4
```

The task `task1` is the sequential composition of the tasks denoted by the expressions $taskexp_1$ and $taskexp_2$. Switching from $taskexp_1$ to $taskexp_2$ occurs when the former terminates, and the composite task `task1` terminates when the latter does. The task `task2` is similar, but the value of the terminating event of the task denoted by $taskexp_3$ is bound to the variable `x` and can thus be used in the expression $taskexp_4$, which denotes the second task.

The following are some of the task-related functions provided by AFRP:

```
mkTask    :: SF a (b, Event c) -> Task a b c
runTask   :: Task a b c -> SF a (Either b c)
runTask_  :: Task a b c -> SF a b
constT    :: b -> Task a b c
snapT     :: Task a b a
timeOut   :: Task a b c -> Time
             -> Task a b (Maybe c)
abortWhen :: Task a b c -> SF a (Event d)
             -> Task a b (Either c d)
```

`mkTask` creates a task from a signal function of the right form. `runTask` turns a task back into a signal function. The output type is `Either b c`, and the output value is `Left` x while the underlying task is running, and becomes the constant value `Right` y, where y is the value of the terminating event, once the task has terminated. `runTask_` is a version of `runTask` where the output becomes undefined after termination. It is typically used for non-terminating tasks. `constT` creates a task with a constant output value. `snapT` is task that terminates immediately. The value of the terminating event is a snapshot of the input at that point in time. `timeOut` adds a time constraint to an existing task by forcibly terminating after the given time interval. In the case of a time out, the value of the terminating event is `Nothing`, otherwise it is `Just` x, where x is the value of the event that terminated the underlying task. Similarly, `abortWhen` adds an extra termination criterion to a task. The value of the terminating event of the resulting task is `Left` x if termination is due to termination of the underlying task, or `Right` y if termination is due to the new criterion, where x and y are the values of the terminating event in question.

4. FROB AND ROBOT PROGRAMMING

Robot programming languages encompasses a broad, disparate collection of work. At one end of the spectrum are languages specifically designed for joint-level or Cartesian motion, e.g. RCCL [7] or Saphira [12]. At the other end of the spectrum are languages that have been motivated by the needs of AI planning or, more generally, *high-level* goal-based specification of behavior, e.g. RPL [14] or PRS [10, 9]. Another set of languages are the so called *intermediate-level* languages which emerged in recent years, e.g. TDL [20], Colbert [11] and Charon [1]. These languages attempt to strike a compromise, offering the ability to program low-level behavior in some detail, while at the same time providing language abstractions that facilitate the description of higher-level system behavior.

Many (indeed nearly all) of these languages have no precise formal specification, and thus can only be analysed anecdotally. However, it is possible to identify a few common themes, such as focus on reactivity (tight coupling of perception to action) and switching between various modes of operation in response to discrete events. Indeed, some of these are arguably concepts intrinsic to the robotics domain. They are also central notions in FRP, embodied in key abstractions such as signal functions, switching combinators, and events. This makes FRP a suitable basis for a robot programming language, and by adding a layer of domain-specific definitions to FRP we arrive at Frob, our language for robot programming.

In conception, Frob most closely resembles an intermediate-level robot programming language. However, since Frob is constructed as a definitional extension around core concepts common to a wide spectrum of robot programming languages, and since it uses a full-fledged functional programming language as the "meta language", it is easy to extend Frob further in the direction of almost any desired robot programming language or paradigm. Thus we can integrate a number of robot programming paradigms addressing different concerns within a single framework. We regard this as a key advantage over most current, less extensible, robot programming languages. Furthermore, this also demonstrates that the thesis of this article is of value not only across application domains, but also within domains.

This section illustrates this point by showing how Frob can be extended to cover some common robot programming paradigms. On a more general note, it also shows the central FRP abstractions in use, and how higher-level abstractions are derived in terms of these, thus paving the way for the next section where we look at multi-domain integration.

4.1 Finite State Automata

Languages like Colbert [11] and Charon [1] employ hybrid systems as their basic execution model. A hybrid system can be thought of as a collection of continuous control modes and a switching logic that governs transitions between these modes. Thus, *Finite State Automata* (FSA) are very convenient for describing the sequences of behaviors which define a complex task to accomplish a high-level goal [2, 11]. In an FSA description, each node represents a continuous mode of operation, and a transition denotes a discrete switch to a new continuous behaviour; see figure 3.

An FRP task can be seen as a simple example of an operating mode associated with a transition. Since FRP tasks form a monad, Haskell's `do`-notation can be used to encode an FSA conveniently, at least as long as the FSA structure is not too complicated (mostly sequencing). For example, the FSA in Figure 3(a) can be coded as follows in Frob. Note that `t2` and `t3` can get "stuck", a condition which we assume is indicated by the return value. If a task gets stuck, the active behavior switches to the node `t4` to handle the situation, and then `t5` which decides whether to restart by calling t recursively, or terminate the overall task.

```
t = do
    t1
    res <- t2
    case res of
        NotStuck -> do
            res <- t3
            case res of
                NotStuck -> t6
                Stuck    -> handleStuck
        Stuck -> handleStuck
where
    handleStuck = do
        t4
        x <- t5
        if x then t else t6
```

Although it is not common in standard FSA formulations, it is highly useful to allow any value associated with a transition to be used to *instantiate* the next mode; see Figure 3(b). This is of course very easy to achieve in our monadic setting, since the transition value is just the return value of the task monad:

```
do
  (x,y) <- findObject
  goto (x,y)
```

Indeed, since tasks and signal functions are first class entities, nothing stops us from *returning* a task or a signal function from a task directly, and then switch into this dynamically instantiated entity later. For example:

```
do
   newTask <- mkTask ..
   newTask
```

We will see this used to good advantage in section 5 for creation and use of visual trackers. But of course, this takes us out of the realm of *finite* state automata.

Figure 3(c) illustrates that this approach to FSA encoding naturally allows for hierarchical automata since composition of tasks yields new tasks. In particular, `t` is a task whose terminating event is given by the subtask `t6`. Thus `t` can be used as a node in a higher-level FSA. Note that the inner structure of composed tasks is not visible from the outside. For example, if new termination criteria, such as a timeout, were to be added to a composite node, all subtasks will be terminated as soon as the termination criteria are fulfilled.

We can also define an FSA abstraction which captures the graph representation of an FSA more closely. We will take the nodes to be *non-terminating* tasks of type `RobotTask ()`:

```
type RobotTask e = Task PioneerInput
                        (MR PioneerOutput) e
```

Transitions are initiated by independent, event-generating signal functions being run in parallel with the node task.

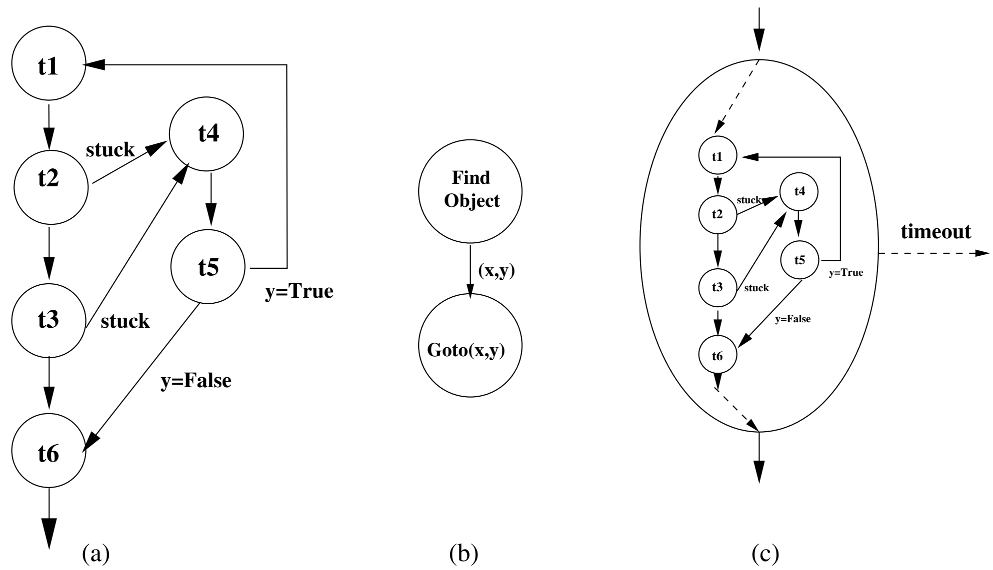

Figure 3: Expressing FSA's in Frob

Nodes are identified by string labels, and each transition is associated with such a label denoting the destination. The point here is to illustrate how a new domain-specific language can be introduced by leveraging the existing concepts. We thus deliberately keep the new abstraction as simple as possible in order to not obscure this point, while acknowledging that one in practice might want an abstraction with more general applicability.

We introduce `TaskNet` as the type of FSA descriptions. Elements of this type are just lists with one element for each node in the FSA giving its label, the associated task, and a list of possible transitions. A transition is just a pair of an event-generating signal function and destination label.

```
type TaskNet =
    [(Label, (RobotTask (), [Transition]))]
type Transition =
    (SF PioneerInput (Event ()), Label)
type Label = String
```

The following recursive function implements a simple interpreter for task nets. Given a label identifying the current state and a task net, it looks up the task for the current state along with its associated transitions. This task will decide the overall behaviour until one of the transitions occur, signalled by an event carrying a label identifying the next state. The new state is used to invoke the interpreter recursively. A label that does not exist in a task net is taken to identify a final state and causes the interpreter to exit with that label as the final result.

```
runTaskNet :: Label -> TaskNet
            -> RobotTask Label
runTaskNet l tn =
    case lookup l tn of
        Nothing       -> return l
        Just (t, trs) -> do
            rl <- t `abortWhen` (joinTrans trs)
            case rl of
                Left _  -> error "Task died!"
                Right l -> runTaskNet l tn
```

Note that the overall result of applying `runTaskNet` to a task net is a task. Thus, this way of describing an FSA integrates elegantly with other task-based FSA encodings. This might seem a trivial point, but the ability to extend the domain vocabulary by capturing design patterns in new abstractions is a big practical advantage over languages that only offer a fixed set of abstractions.

The utility function `joinTrans` turns a list of transitions into a single event-generating signal-function:

```
joinTrans :: [Transition]
             -> SF PioneerInput (Event Label)
joinTrans [] = never
joinTrans ((e,l):rest) =
    let e1 = e `tagSF` s
        e2 = joinTrans rest
    in (e1 &&& e2) >>> (arr (uncurry lMerge))
```

The example below shows how the task net abstraction can be used. In the resulting task, the user can change the robot's behavior by using the mouse buttons. A left click will change the type of the tracker being used, while a right click will toggle the obstacle avoidance behavior between disparity based obstacle avoidance and sensor based obstacle avoidance.

```
net1 =
  [ ("S0",(nullT,[(lbpE,"S1"),(rbpE,"S3")])),
    ("S1",(task1,[(lbpE,"S2"),(rbpE,"S3")])),
    ("S2",(task2,[(lbpE,"S1"),(rbpE,"S4")])),
    ("S3",(task3,[(lbpE,"S4"),(rbpE,"S1")])),
    ("S4",(task4,[(lbpE,"S3"),(rbpE,"S2")]))
  ]
  where
    task1 = colorTracking `withOA` disparityOA
    task2 = ssdTracking   `withOA` disparityOA
    task3 = colorTracking `withOA` sensorOA
    task4 = ssdTracking   `withOA` sensorOA
```

4.2 Subsumption Architecture

The subsumption architecture is perhaps one of the most widely cited ideas for programming reactive robot systems [3, 4]. A subsumption "program" is a set of blocks (signal functions) connected by "wires" along which information flows. Programs are designed in layers, ranging from simple low-level behaviors to high-level complex behaviors. The coordination process for defining complex behaviors from simpler ones only depends on two basic mechanisms: inhibition and suppression. Inhibition means a signal is prohibited reaching the actuators and suppression means the signal is replaced with another suppressing message.

We can implement something similar in Frob by associating priorities with signals. We introduce the type synonym `Stimulus` for a "prioritized" signal sample, along with some convenient abbreviations:

```
type Priority   = Double
type Stimulus a = (a, Priority)
type Behavior a = SF PioneerInput a
type RControl   = MR PioneerOutput
```

A prioritized behavior can easily be derived from an unprioritized behavior and a priority-determining behavior:

```
withPriority ::
    Behavior a -> Behavior Priority
    -> Behavior (Stimulus a)
withPriority b p = b &&& p
```

Subsumption between prioritized behaviors is captured by the following definition. It is a simple pointwise selection of the stimulus with the highest priority:

```
subsumes ::
    Behavior (Stimulus a)
    -> Behavior (Stimulus a)
    -> Behavior (Stimulus a)
subsumes b1 b2 = (b1 &&& b2) >>> (arr f)
    where
        f ((a1,p1),(a2,p2)) =
            if p1>p2 then (a1,p1) else (a2,p2)
```

Let us outline an example showing how these can be used. We want to define a robot task where the robot will wander randomly while avoiding obstacles, pick up pre-designated objects should they happen to be close, and travel home once all pre-designated objects have been picked up.

The homing behavior is parameterized on a boolean behavior indicating if it is time to return home. If not, then the homing priority is 0 (i.e. the behavior is not active), otherwise the priority is 2.

```
homing :: Behavior Bool
       -> Behavior (Stimulus RControl)
homing allDone =
    (travelTo home)
    `withPriority` (allDone >>> arr f)
    where f p = if p then 2 else 0

travelTo :: Point -> Behavior RControl
travelTo p = ...
```

Pickup is initialized with a set of points representing the positions of the pre-designated objects to pick up. Its output is a prioritized control signal paired with a boolean signal indicating whether all pre-designated objects have been picked up. The pickup behavior's priority is 0 until the first object is close. Then it raises to a sufficiently high priority, say 5, until the object has been picked up, at which point the behavior will invoke itself recursively on the rest of the object position list.

```
pickup :: [Point]
       -> Behavior (Stimulus RControl,Bool)
pickup [] = (idle `withPriority` constant 0)
            &&& (constant True)
pickup (p:ps) = ...
```

The wandering behavior has a constant priority of 1. The priority of the avoidance behavior is determined by sonar readings. If an obstacle is sufficiently close, the priority is raised so that avoidance suppresses all other navigation tasks. Once an obstacle has been successfully avoided, the priority of avoidance will drop and the other tasks regain control. The details are omitted.

```
wandering :: Behavior (Stimulus RControl)
wandering = randomWalk `withPriority` (constant 1)

avoiding :: Behavior (Stimulus RControl)
avoiding = ...
```

Finally we can combine these prioritized behaviors using the subsumption operator. Note how `pickup` is used to create *two* behaviours, one for picking up objects, one for indicating when all objects have been picked up. The latter is passed to `homing` where it controls the priority of the homing behaviour, as described above.

```
system ::  Behavior RControl
system = (homing allDone
         `subsumes` pickup'
         `subsumes` wandering
         `subsumes` avoiding)
 >>> (arr fst)
    where
        p = pickup [(200, 200), (100, 100)]
        allDone = p >>> (arr snd)
        pickup' = p >>> (arr fst)
```

5. A TRACKING-BASED NAVIGATION SYSTEM

This section illustrates how notions from the application domains of robotics, vision, and user interfaces can be integrated within the AFRP framework. We illustrate through a simple but still realistic system for vision-based robot navigation. The task of the robot is to follow an object while avoiding bumping into obstacles. The user can interact with the system by choosing what object to track and follow, which kind of tracker to use, toggling the obstacle avoidance behavior on and off, and, if necessary, switch to manual control.

We first turn our attention to visual tracking. Trackers typically maintain a state that describes the position of the tracked feature in the video frame. Under the assumption that the tracked feature has not moved too far between two consecutive video frames, the state is used to acquire an image region from the latest video frame covering the tracked feature. A *stepper* is then used to compute the new state, and the feedback loop is closed by feeding this to the acquiring component. The stepper may also return an accuracy value which can be used to estimate the target position more correctly. Since only a part of the video frame is used for the computation, stateful tracking is more efficient than processing the whole image as long as the tracked feature does not move too quickly.

The two main components of a tracker are thus the *source*, which acquires the target subimage, and the stepper, which computes the new state. In our setting, both are signal functions. The source takes the whole image signal and produces the subimage signal, along with the state information used for acquiring the subimage. The stepper consumes the signal from the source and produces a signal of state values. The `init` parameter in the stepper type specifies an initialization value for the stepper. For instance, this can be a color value for a color tracker. XVision2 stepper functions generally return a blob, which is a rectangular region described by the upper left corner coordinates and width and height of the blob.

```
type Src state =
    SF (ImageRGB, state) (ImageRGB, state)

type Stepper init state =
    init -> SF (ImageRGB, state) state

type Blob = (Int, Int, Int, Int)
```

The following is a very simple source function that just returns the subimage:

```
source1 :: Src Blob
source1 = proc (im, blob) -> do
    returnA -< (subImage (im, blob), blob)
```

Next we define a stepper that uses color for identifying the feature to track:

```
colorStepper :: Stepper ColorSelector Blob
colorStepper c = arr f
    where
        f (im, blob1) = g blob1
                          (stepBlob im c)
        g b1 b2 = let (x, y, w, h)  = b1
                      (x2,y2,w2,h2) = b2
                      pad = 10
                  in limitB $
                     if w2<=0 || h2<=0 then b1
                     else (x+x2-pad, y+y2-pad,
                           w2+2*pad, h2+2*pad)
```

The XVision2 (native C++) function `stepBlob` is called first. If no blob is detected for the given color value (which sometimes happens, for example due to light conditions), the state remains the same. Otherwise the new state is computed with some padding around the region to provide a safety margin. `limitB` is used to limit the coordinates according to the screen dimensions. We then combine the source and the stepper in a feedback loop to define our tracker:

```
makeTracker :: state -> init -> Src state
               -> Stepper init state
               -> SF ImageRGB state
makeTracker s0 i src stp = proc img -> do rec
    (subImg, s1) <- src     -< (img, s)
    s'           <- stp i   -< (subImg, s1)
    s            <- iPre s0 -< s'
    returnA -< s
```

`makeTracker` can be used to define a tracker with any source or stepper as long as they share the same state type. It returns a signal function that takes whole images and returns the tracking state. Note how the initialized delay operator `iPre` is used to close the loop.

The next step is to define a pair of tasks for interactive creation of trackers :

```
getBlob :: RobotTask Blob
getBlob = do
   c1 <- getClick
   let sf = proc inp -> do
                mpos <- mousePos -< inp
                returnA -< drawRect c1 mpos
   c2 <- mkTask (sf &&& fvgiLBP)
   let (x1,y1) = c1
       (x2,y2) = c2
   return (x1,y1,(x2-x1),(y2-y1))

colorTracker :: RobotTask (SF ImageRGB Blob)
colorTracker = do
   blob0 <- getBlob
   inp   <- snapT
   let c = getColorSelector
             (subImage ((fviImage inp),blob0))
       track = makeTracker blob0 c source1
                               colorStepper
   return track
```

In `getBlob`, the user selects the first corner of the blob by clicking the left mouse button. The corner returned by `getClick` (`c1`) is then used to define a signal function (`sf`) which outputs a rectangle from this corner to the current mouse position. This signal function is used to form a task which terminates on a left mouse button click, returning the current position as the second corner of the blob (`c2`). In `colorTracker`, a snapshot is taken of the current input once a blob has been selected since the color to be tracked inside the blob must be determined on the current frame. The color of the blob is then determined and used for creating a color-based tracker. Note that the first class status of signal functions (here, the tracker) is crucial in this definition and where it is used below, especially the abilty to *dynamically* instantiate and then switch into a signal function.

Up to this point, we have used tasks for the vision component of the system, but no commands have been sent to the robot. Thus we see that the task abstraction is useful in domains beside pure robot programming.

Let us now use the tracker in code for driving the robot and for displaying a rectangle on the live image showing the region being tracked.

```
drive = do
  tr <- colorTracker
  let sf = proc inp -> do
    (x,y,w,h) <- tr -< fviImage inp
    (v,t) <- arr driveVector -< toVector
                                  (mid(x,y,w,h))
    returnA -< ddVelTR v t `mrMerge`
                drawBlob (x,y,w,h)
  res <- mkTask (sf &&& fvgiLBP)
         `abortWhen` fvgiRBP
  case res of
    Left _  -> drive
    Right _ -> followMouse
```

```
driveVector :: Vector -> (Velocity, RotVel)
driveVector (x,y) = (v,w)
  where
    a = radian x y
    v = limitA maxVelocity (k1 * x)
    w = limitA maxTurn (signum a * k2 * a * a)
```

The image part of the input signal selected by `fviInput` is fed into the tracker to get the blob state. The function `toVector` takes a screen coordinate, which is in this case the middle of the blob, and produces a force vector which conceptually pushes the robot in the desired direction. Since the camera is mounted on top of the robot and showing what is ahead of the robot, this vector can be computed from the middle of the bottom of the screen and the screen coordinate. The resulting vector indicates whether the target is to the left or right of the robot. How much to the left or right determines the turning velocity. The magnitude of the vector is used to determine the translational velocity. The conversion of the imaginary force vector into appropriate rotational and translational velocities is done by `driveVector` where both velocities are bounded for safety reasons.

The signal function `sf` both drives the robot, by feeding the translational and rotational velocities to the command function `ddVelTR`, and draws a rectangle at the blob. This signal function is used to define a task which either terminates normally with a left button click or, because of the added termination criterion, with a right button click. For the former case, task is initiated again enabling user to select another target to track. Alternatively if the task terminates with a right click then the overall task switches to the task `followMouse`, where the user can drive the robot with the mouse. The definition of `followMouse` is very similar to `drive`, except that the force vector is computed from the mouse position this time instead of blob.

Figure 4 shows an extension of the example. It adds a simple user interface, more user interaction and obstacle avoidance. The signal function `butSG` draws three buttons, i.e. rectangles with labels inside. The event source `butE` generates events corresponding to left button clicks inside a rectangle, with the event value specifying which button is pressed. The resulting events are merged into one by `buttonE`. This final event will be used as the terminating event of the driving task to choose between different continuations. The two first buttons allow the user to switch to a new target, specifying that either color tracking or disparity tracking should be used. The third button switches to manual control.

The right button is used to toggle the obstacle avoidance behavior on and off. The signal function `accumHold` takes an event input signal where the events carry functions of type `a -> a`. It maintains an internal state initialized to the supplied initial value. The value of this state is also the value of the continuous output signal. Whenever an event occurs, the function carried by the event is used to compute a new internal state. This means that `toggleOA` is a boolean signal changing between `True` and `False` every time a right button event occurs. `toggleOA` is both used to turn the obstacle avoidance behavior on and off, and to display the current obstacle avoidance mode on the screen in `obsavSG`.

The XVision2 function `obsLines` returns a list of lines which are projections of obstacles ahead. These lines are used by `toVector2` with the position of the tracked object for computing a force vector for guiding the robot. If there is no obstacle right in front of the robot, it goes forward by following the force vector produced by the tracked object. If there is an obstacle in front, the closest gap between the obstacles that is large enough for the robot to pass through is found, and another force vector is computed towards this gap. Since the magnitude of the obstacle avoidance vector will increase the closer an obstacle is, this vector will eventually dominate over the tracking vector, causing the robot to pass through the gap. After the obstacle has been cleared, the tracking vector will be dominant again and the robot continues to drive towards the tracked object. The third boolean argument of `toVector2` is used to turn obstacle avoidance on and off.

Figure 5: User-interface while robot is navigating towards the tracked object

Figure 5 shows the user interface in a frame coming from the stereo camera mounted on top of the robot while the robot is driven by `drive2`.

6. CONCLUSIONS

To date, robot programming has been an area rife with innovative approaches to systems and software development but relatively lacking in principled analysis. Yet, it is an area that provides an interesting and challenging basis for programming language research. In particular, the fact that almost all operations are based on time-varying quantities (indeed, time is a *critical* factor) in most algorithms offers new and interesting challenges to programming languages.

Our experience with FRP has been quite promising. We have found it to be a flexible and powerful method for integrating the various time-based computations required in vision, robotics, graphics, and human-machine interaction. Furthermore, the recent development of AFRP has been a great practical step forward in terms of the scalability of our systems.

At the same time, we have learned several lessons in the use of FRP. First, we have found that most toolkits cannot be treated as black boxes with a single, high level interface. For example, our vision code is imported at a level which allows us to redefine some of the underlying C++ abstractions (e.g. a visual tracker) in FRP. This level of interface is far more flexible and useful when integration occurs. Second, we have found that there is often a natural evolution from high-level (FRP) code to low-level (e.g. C++) code as applications develop and tool-box deficiencies appear. As a

```
data Navigation_Mode = ColorTracking | DisparityTracking | Manual

but1, but2, but3 :: Blob
but1 = (10, 10, 90, 20) -- similar for but2 and but3

butE :: Blob -> a -> SF PioneerInput (Event a)
butE rect res= fvgiLBP >>> (arr check)
    where
        f (x0, y0, w, h) (x, y) = x>=x0 && x<=x0+w && y>=y0 && y<=y0+h
        check ev = (filterE (f rect) ev) `tag` res

but1E, but2E, but3E :: SF PioneerInput (Event Navigation_Mode)
but1E = butE but1 ColorTracking
but2E = butE but2 DisparityTracking
but3E = butE but3 Manual

buttonE :: SF PioneerInput (Event Navigation_Mode)
buttonE = proc inp -> do
    (e1,(e2,e3)) <- (but1E &&& but2E &&& but3E) -< inp
    returnA -< lMerge e1 (lMerge e2 e3)

butSG = drawBlob but1 `mrMerge` drawBlob but2 `mrMerge` drawBlob but3
        `mrMerge` fvgoOverlayText (20, 25) "COLOR TRACKER" Red
        `mrMerge` fvgoOverlayText (15, 55) "STEREO TRACKER" Red
        `mrMerge` fvgoOverlayText (40, 85) "MANUAL" Red

drive2 trackTask = do
    tr <- trackTask
    let sf = proc inp -> do
                 rbpE <- fvgiRBP -< inp
                 toggleOA <- accumHold False -< rbpE `tag` not
                 (x,y,w,h) <- tr -< fviImage inp
                 ls <- obsLines  -< fviImage inp
                 (v,t) <- arr driveVector
                          -< toVector2 ls (mid (x,y,w,h)) toggleOA
                 returnA -< ddVelTR v t
                            `mrMerge` drawBlob (x,y,w,h)
                            `mrMerge` butSG
                            `mrMerge` obsavSG toggleOA
    res <- mkTask (sf &&& buttonE)
    case res of
        ColorTracking     -> drive2 colorTracker
        DisparityTracking -> drive2 dispTracker
        Manual            -> followMouse2
    where
        obsavSG p = let str = if p then "Obstacle Avoidance :: ON "
                              else "Obstacle Avoidance :: OFF"
                    in fvgoOverlayText (160, 20) str Red
```

Figure 4: Main task for Tracking-based Navigation System

result, there is a natural merging of functionality over the life of a project. Finally, we have found that working in FRP is ideal for the prototyping "beyond the state of the art." The flexible facilities of FRP (and its embedding in Haskell) make it an ideal platform to discover and develop domain abstractions.

7. ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the NSF under grant EIA-9996430 and the DARPA MARS program.

8. ADDITIONAL AUTHORS

Additional authors: Darius Burschka (Johns Hopkins University, email: `burschka@cs.jhu.edu`) and John Peterson Yale University, email: `peterson@cs.yale.edu`).

9. REFERENCES

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.

[2] R. C. Arkin. *Behavior-based Robotics*. MIT Press, 1998.

[3] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, April 1986.

[4] R. A. Brooks. Integrated systems based on behaviors. *SIGART Bulletin*, 2(4):46–50, 1991.

[5] G. Hager and J. Peterson. A transformational approach to the design of robot software. In *Robotics Research: The Ninth International Symposium*, pages 257–264. Springer Verlag, 2000.

[6] 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, Jan. 1998.

[7] V. Hayward and . R. P. Paul. Robot manipulator control under unix RCCL: a robot control "C" library. *The International Journal of Robotics Research, Winter 1986*, 5(4):94–111, 1986.

[8] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[9] F. Ingrand, R. Chatila, R. Alami, and F. Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Proc. of the IEEE International Conf. on Robotics and Automation*, Minneaplois, USA, 1996.

[10] F. Ingrand, M. Georgeff, and A. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.

[11] K. Konolige. Colbert : A language for reactive control in sapphira. Technical report, SRI International, June 1997.

[12] K. Konolige, K. Myers, and E. Ruspini. The saphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9, 1997.

[13] S. Lang. The xvision2 software system, May 2001. Master Thesis, Dept. of Computer Science, The Johns Hopkins University.

[14] D. McDermott. A reactive plan language. Technical report, Yale University Department of Computer Science, 1991.

[15] H. Nilsson, A. Courtney, and J. Peterson. Functional Reactive Programming, continued. Submitted for publication, June 2002.

[16] R. Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.

[17] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conf. on Robotics and Automation*, May 1999.

[18] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with Haskell. In *Proc. 1st International Conference on Practical Aspects of Declarative Languages (PADL'99)*, pages 91–105, January 1999.

[19] A. Reid, J. Peterson, P. Hudak, and G. Hager. Prototyping real-time vision systems: An experiment in DSL design. In *Proc. 21st International Conference on Software Engineering (ICSE'99)*, May 1999.

[20] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*, 1998.

[21] Z. Wan and P. Hudak. Functional Reactive Programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, Vancouver, British Columbia, Canada, June 2000.