A Comparative Review of Robot Programming Languages

Izzet Pembeci, Gregory Hager

August 14, 2001

Abstract

In this paper, we make a comparative review of a variety of "intermediate-level" robot languages that have emerged in recent years. We also describe a robot programming language called FROB (for Functional ROBotics). FROB is an example of an embedded, domain-specific language, hosted by the Haskell programming language.

We present the basic concepts of the language, discuss the ability of FROB to model other programming architectures, and we compare it's capabilities to other extant languages that are reviewed.

1 Introduction

The appropriate languages, language structures, and software architectures for developing robot software has been the topic of debate and discussion since the earliest, computer-controlled robot systems. The fact that no uniform consensus (at least in terms of a "universal" robot programming system) is not surprising, given the range of issues such a language would need to address.

One set of issues relates to the simple fact that an autonomous robot must inhabit an unstructured, dynamic, changing world, and therefore must be able to effectively react to that world using sensor information and physical effectors. At a very basic level, this implies that the language needs to facilitate the description of sensor-based control algorithms. However, unpredictability implies that any such algorithm might be interrupted at any time, or may suddenly be faced with inputs that are inconsistent or incomprehensible with the current control context. Methods such as prioritization of control [2] or monitors [15] have been used in the past to deal with such issues. Ideally, a good robot programming language should provide a simple, clean set of abstractions for describing and combining control behaviors.

A second set of issues arises from the nature of the tasks that autonomous robots would (ideally) perform. Many of these tasks are easily specified at an abstract level (e.g. deliver the mail, give a tour, or collect these objects), but their execution ultimately entails a complex interaction and sequencing of low-level behaviors. This has led to the notion of a layered system [24] [23], where there is a logical separation between high-level planning, mid-level execution and monitoring of the steps of a tasks, and low-level execution of a set of behaviors. Ideally, a robot programming language should span these layers, *thereby providing a uniform means for creating abstractions and relating them to one another.*

Finally, a third set of issues arises due to the practical problems of developing robot system software. One fundamental fact is that robot system programming is highly experimental in nature: we often do not know at the outset how best to use sensors, control actuators, set thresholds, and so forth. Thus, there is often a long cycle of development, testing, redevelopment, and refinement until a system reaches the point where it operates reliably. As a result, rapid, correct prototyping and software reuse are of paramount importance. Likewise, the ability to combine many system components quickly and correctly is essential. A closely related

issue is the desire to develop "hardware independent" algorithms, so that code may be easily transferred from application to application.

A more subtle practical problem is the fact that, at any given time, the computation needed to allow a robot to achieve its objectives is highly dynamic. More precisely, from one time interval to the next, there may be a large shift in the sensors employed, the algorithms used to process that information, the control algorithms currently operating, and the surrounding system monitoring that may be taking place. A good language should spare the programmer the details of explictly managing the flow of computation, while keeping code execution to the minimum necessary to perform the currently active computations.

In this article, we introduce a new robot programming language called FROB (Functional ROBotics). FROB was motivated, in part, by a desire to create a "broad spectrum" approach to robot programming — that is, an approach that can be readily adapted to the needs and requirements of all types of robot programming problems. FROB achieves goal by taking advantage of the fact that it is an *embedded* language hosted by a lazy, higher-order, strongly typed language called Haskell [21]. As such, it is possible to express many different programming architectures within the same system. At the same time, FROB makes use of a lazy execution model, thereby achieving the aims of minimizing computation.

In the remainder of this article, we describe FROB and perform a comparative analysis of its capabilities. In the next section, we review several recent programming systems and discuss how they address the issues outlined above. We then describe FROB and compare its capabilities to these languages using a set of examples. Finally, we conclude with a discussion of current and future directions of the FROB project.

2 A Review of Programming Languages

The term "robot programming language" encompasses a broad, disparate collection of work. At one end of the spectrum are languages[1] specifically designed for joint-level or Cartesian motion, e.g. RCCL [10] or Saphira [16]. 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 or PRS.

In this review, we will generally focus to a variety of "intermediate-level" languages have emerged in recent years, e.g. TDL [24] or Colbert [15]. 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. Although our review is focussed on these systems, we also discuss a variety of other architectures in Section 3.

2.1 Colbert

Colbert is called as a sequencer language by its developers [15]. It is part of the Saphira architecture [16]. The Saphira architecture is an integrated sensing and control system for robotics applications. Complex operations like visual tracking of humans, coordination of motor controls, planning are integrated in the architecture using the concepts of coordination of behavior, coherence of modeling, and communication with other agents. The motion control layer of Saphira consists of a fuzzy controller and Colbert is used for the middle execution level between the motion control layer and planning.

Colbert programs are activities whose semantics is based on FSAs and are written in a subset of ANSI C. The behavior of the robot is controlled by activities like :

[1]Here and in the following text, we will use the term "language" to broadly denote all types of programming systems, from embedded languages using libraries to full-blown architectures with complete, stand-alone programming languages.

- Sequencing the basic actions that the robot will perfom.
- Monitoring the execution of basic actions and other activities.
- Executing activity subroutines.
- Checking and setting the values of internal variables.

Robot control in Colbert means defining such activities as activity schema each of which corresponds to a finite state automaton. The activity executive interprets the statements in an activity schema according to the associated FSA. The statements of a schema do not correspond directly to the states of the FSA. For instance, conditional and looping statements will be probably represented as a set of nodes.

Actions at the nodes are typically primitive robot actions and internal state changes. When an activity is finished, a transition is done to a new node according to the success or failure of that activity. There is an implicit wait condition, a transition to itself, in the nodes but this can be overwritten to have concurrent activities. The action executive updates concurrent activities in a round-robin fashion. In each clock cycle, every executing activity will progress through at least one state of its associated FSA.

Communication between activities is possible directly by sending signals, or indirectly by accessing and modifying a global database consisting of internal variables. There are some predefined states of an activity which can be checked or changed by other activities. Such states include :

- initial state of an activity.
- termination states (success, failure or timeout)
- suspend or interrupted states
- resumed state

These special states enable activities to signal other activities (possibly including some subactivities spawned by the parent) for coordination or for handling dynamic changes in the environment. It is also possible to define an activity's life span depending on other activities by querying these predefined states.

To summarize, when an activity is executed, it can either invoke new activities or result in some primitive actions (i.e commands send to the robot). Activities can communicate and affect each other. The internal database is used for storing related information coming from the sensors (but this part is not Colbert's responsibility in the Sapphira architecture) and also to enable activities to share information.

Figure 1 shows an approach activity example from Colbert. In the first simple `patrol` example the robot moves back and forth n times between two points 1 meter apart. There will be a FSM corresponding to this activity and the activity's name, `patrol`, can be referred in others. `turnto` and `move` are built-in primitive robot actions.

In the `approach` activity the robot patrols for a specified amount of time. While patrolling if it detects an object at less than 2 meters, it moves to within 20 cm. of the object. The subtask `patrol` is spawned with an additional timeout constraint. `ObjInFront` is part of the global database. Its distance is checked in a loop, and if it is further then 2 meters the loop restarts by checking the sub activity patrol, and checking whether a global error occured. Otherwise the loop terminates, suspends the patrol subactivity and the robot moves to the object.

```
act patrol(int n)
{
   while (n!=0)
   { n=n-1;
      turnto(180);
      move(1000);
      turnto(0);
      move(1000): }
}

act approach()
{
   int x;
   start patrol(-1) timeout 300 noblock;
  checking :
   if (timedout(patrol) || sfStalledMotor(sfLEFT))
     fail;
   x = ObjInFront();
   if (x>2000) goto checking;
   suspend patrol;
   move (x-200);
   succeed;
}
```

Figure 1: A Colbert Approach Activity

In figure 2, `patrol2` defines an activity where the robot patrols n times but patrolling may be interrupted by other activities. In that case the robot waits for any forward motion finished (by the help of `sfDonePosition`) and then stops. When the activity is resumed the robot continues patrolling by going to the code point (node of the FSM) marked by `start`.

2.2 Animate Agent Architecture (Firby)

The Animate Agent Architecture aims to building an agent control system that integrates reactive plan execution, behavioral control, and active vision within a single software framework [6]. The basic datatype of the system are RAPs which are reactive plans defined in a Lisp-like syntax.

At the low level there are soft real-time routines which are called skills. Skills are both used to get information from the world and to effect the world. Skills can communicate with each other and the RAPs by asynchrous signals carried through global channels.

The RAP system's responsibility is to get a set of task goals, which can be a plan from the planner or a top-level goal and then to expand each goal seperately until primitive actions are reached. These primitive actions can disable or enable the skills. Each RAP can generate a set of concurrent skills which will help to reach the goal associated with the RAP and then waits for the skills' success or failure signals.

Some of the basic clauses and methods in a RAP are :

- tests about some condition, queries of the memory. These may be used as predefined conditions to determine whether to execute the RAP in the first place or testing success and failure.

```
act patrol2(int n)
{
  start:
   while (n!=0)
   { n=n-1;
      turnto(180);
      move(1000);
      turnto(0);
      move(1000): }
   succeed;
  onInterrupt:
   waitfor (sfDonePosition());
   suspend;
  onResume:
   n=n+1;
   goto start;
}
```

Figure 2: A Colbert Patrol Activity

- the tasks in a RAP (called task-net) can be ordered sequentially or in parallel and they can even be unordered. The life-span of subtasks can be defined depending on others.
- monitors are used to synchronize RAP execution with changes in the world. The execution can be suspended until some state becomes true, until some event happens, or for a specific period of time by using the monitors.
- it is possible to have different methods to execute depending on different contexts (i.e state of the world or other on going tasks).
- estimated execution time can be given to help the RAP interpreter to satisfy task deadlines.
- the execution time of a RAP can be limited by adding a timeout clause.
- internal variables.

Since there is a mechanism (`wait-for` clause) for defining the starting time of a task relative to the success of another one, the same mechanism can be used to handle exceptions. In fact a task sends a specific signal via the global channels when it succeeds, not just a success signal. So a specific signal describing the exception can be sent similarly and handled by a responsible RAP. For example:

```
(task-net
  (t0 (camera-on (wait-for :success t1) (for t2))
  (t1 (approach-target ?target)
      (wait-for (at-target) t3)
      (wait-for (stuck) t3)
      (until-start t3))
  (t2 (track-target ?target)
      (wait-for (lost-target) t3)
      (wait-for (camera-problem) :terminate)
      (until-start t3))
  (t3 (camera-off)))
```

```
(define-rap (move-object ?container)
 ...
 (on-event (lost-object) :fail
   (sequence
      (t1 (pickup-object))
      (t2 (go-to ?container))
      (t3 (drop-off-object)))))

(define-rap (pickup-object)
 ...
 (sequence
    (t1 (start-pickup))
    (t2 (grasp))
    (t3 (spawn (monitor-hand ?task))
        (mem-add (monitoring-hand ?task))
    (t4 (finish-pickup))))

(define-rap (drop-off-object)
 ...
 (method
    (context (monitor-hand ?task))
    (task-net
       (sequence
          (t1 (start-drop-off))
          (t2 (ungrasp))
          (t3 (terminate ?task))
          (t4 (finish-drop-off)))))
 (method
    (context (not (monitor-hand ?task)))
    (task-net
       (sequence
          (t1 (start-drop-off))
          (t2 (ungrasp))
          (t4 (finish-drop-off))))))
```

Figure 3: RAP Example

This task net defines a task where a target is approached and simultaneously tracked. The camera is turned on at the start of the task and turned off at the task completion or some error occurs. Firby gives this example in [7] to illustrate how the different subtasks can be sequenced and how the result of a task can result in different continuations. `wait-for` checks for the asynchrous signals carried through global channels. `until-start` determines whether a task is started.

Firby gives the example in figure 3 to show that spawning independent tasks can be useful not to give up modularity. Note how the `monitor-hand` task is spawned in the subtask `pickup-object` and added to the memory and then terminated in the subtask `drop-off-object`. `context` is used to check a condition to determine which methods will be activated.

2.3 Behavior Language

Brooks is one of the first advocates of reactive behavior-based methods for robot programming. So not surprisingly, his subsumption architecture is based on different layers, each of which work concurrently and asynchronously to achieve individual goals [2] [3]. In the earlier designs, the behaviors were represented by augmented finite state machines (AFSMs), thus the Behavior Language still includes AFSMs as the low level building blocks. The Behavior Language has a Lisp-like syntax and compilers available even into programmable-array logic circuits.

An AFSM encapsulates a behavioral transformation function where the input to the function can be suppressed or the output can be inhibited by other components of the system. It is also possible to reset an AFSM to its initial state. Each layer in the subsumption architecture has a specific goal. The higher layers can use the output of the lower levels and also affect their input and output to achieve their goals which are generally more abstract then the goals of the lower layers. It is argued that this kind of hierarchical interaction between layers prohibits designing higher levels independently.

When an AFSM is started, it waits for a specified triggering event and then its body is executed. Such events can depend on time (i.e. periodically activate the AFSM), a predicate about the state of the system, a message deposited to a specified internal register, or other components' being enabled or disabled. In the body it is possible to perform primitive actions or to put messages in order to interact with other AFSMs.

In the Behavior Language behaviors are represented by a set of rules which are compiled to AFSM representations and these can be compiled for the target processors. By grouping AFSMs into behaviors, it is possible to share registers, outputs, monitoring actions and more importantly have a more abstract component. It is possible to explicitly connect isolated ASFMs and behaviors together by using the `connect` clause. These connect clauses are also used to suppress an input port or to inhibit an output port. There are no components like plans or procedures but it is possible to define macros and use them in the definition of behaviors.

2.4 PRS

Procedural Reasoning System (PRS) is a general framework designed as a so-called situated reasoning system [14]. Such systems are used for diagnosing and taking necessary measures to handle plant and process malfunctions in real time. This requires reasoning about management of tasks which includes reasoning about the criticality or urgency of tasks, potential interactions between tasks, the execution order of tasks, the need for resuming and suspending tasks depending on the state of the system, and finally which tasks to execute to reach the goals of other tasks. As argued in [13] PRS can be also adapted to robot control. Plans can be defined in a Lisp-like language or by using a graphical tool.

The basic elements of the system are :

- a database containing the system's current beliefs about the world. These are automatically updated as new events appear. It is also possible to compute values on demand.
- a library of plans (or procedures, or scripts) which describe a particular sequence of actions and tests that may be performed to achieve given goals or to react to certain situations. These plans are also called Knowledge Areas and they are application dependent.
- a task graph which is a dynamical set of tasks currently executing.

The PRS interpreter checks new goals and new events which can be triggered by the outside world or by

active tasks, selects appropriate plans to handle these new goals and events based on the database, places these selected procedures to the task graph and then finally executes one step of the active procedure. This can result in a command about the real world or a new goal.

A plan generally includes :

- `achieve goal` statements, which will add new tasks to the task graph to satisfy this new subgoal,
- tests for some condition, iterations and possible continuations depending on tests,
- waiting for some condition to become true,
- preserving some conditions, which are used as guarding actions (i.e. adding a constraint to a task. If the constraint fails then the action is suspended),
- maintaining some criteria (i.e. `maintain battery-level 0.2` will result in checking battery level during the task, interrupting the goal if the battery level falls under 0.2 and trying to reestablish it to the desired level and then returning to the interrupted task(s)).

There is a difference in how the PRS interpreter adds new procedures depending on the new goals and new events. For a goal, each unifiable procedure is tried one after another, so each procedure has a context (a test depending on the state of the system) to run. The goal is declared fail if all the related procedures fail. In the notion of event, the invoked procedures do not pursue an explicit goal, e.g. a monitor. So they are like a response to an event and their success or failure is not analyzed.

2.5 TDL

As discussed earlier, the robot control architectures can be developed as 3 interacting layers. The behavior level interacts with the physical world. The planning layer is used for defining how to achieve goals. The executible layer connects these two layers issuing commands to the behavior level which are results of the plans and passing sensory data taken from the behavior level to the planning layer to enable planning reactive to the real world. So the executive layer is responsible for expanding abstract goals into low-level commands, executing them and handling exceptions.

The main motivation behind developing Task Description Language (TDL) [24] is that using conventional programming languages for defining such task-level control functions results in highly non-linear code which is also difficult to understand, debug and maintain. TDL extends C++ with a syntactic support for task-level control. A compiler is available to translate TDL code into C++ code that will use the Task Control Management (TCM) libraries.

The basic datatype of TDL is the task tree. The leaves of a task tree are generally commands which will perform some physical action in the world. Other types of nodes are goals, representing higher level tasks, monitors and exceptions. An action associated with such nodes can perform computations, and change the structure of the task tree (i.e. goals will add child nodes to the tree which can be viewed as subgoals). The nodes of a task tree can be executed sequentially or in parallel. It is also possible to expand a subtree but wait for some synchronization constraints to hold before beginning executing it. This is achieved by having different modes for nodes which are called the state of the node. There is a well defined semantics for expanding and executing nodes of a task tree. Briefly a node is disabled when there are synchronization constraints that are not satisfied, enabled otherwise, active if it is enabled and there are sufficient resources (both computational and physical), and finally completed if the action related to that node succeeds or fails.

```
Goal deliverMail(int room)
{
  double x,y;
  getRoomCoordinates(room, &x, &y);
  spawn navigateToLocn(x, y);
  spawn centerOnDoor(x,y)
    with sequential execution previous,
    terminate in 0:0:30.0;
  spawn speak("Xavier here with your mail.")
    with sequential execution centerOnDoor,
    terminate at monitorPickup completed;
  spawn monitorPickUp()
    with sequential execution centerOnDoor;
}

Goal centerOnDoor(double x, double y)
  delay expansion
{
  int whichSide;
  spawn lookforDoor(&whichSide) with wait;
  if (whichSide !=0) {
     if (whichSide<0)
        spawn move(-10); // move left
     else
        spawn move(10); // move right
     spawn centerOnDoor(x,y)
       with disable execution until
         previous execution completed;
  }
}
```

Figure 4: A Task Tree for Mail Delivery

A goal's termination or disabling criteria may be defined relative to time or starting or finishing time of an event which is typically triggered by other goals' termination, expansion, or starting to execute. Explicit labels may be used to differentiate multiple spawning of the same task. Monitors have the same structure of goals with some additional implicit constraints like max trigger, max activations or period of activity. Exceptions can be thrown when a goal fails and they can be handled in the goal body by spawning appropriate exception handling tasks.

The goal definitions in figure 4 will result in a task tree where the robot will goto a specific location to deliver mail. Note how the nodes are defined to be executed sequentially or the expansion of the tree and execution is delayed in `centerOnDoor` since it is recursively spawns itself until the robot has a position at the center of the door. The main goal definition will terminate `centerOnDoor` after a specific amount of time, namely 30 seconds.

2.6 SIGNAL

SIGNAL is a language designed for safe real-time system programming [8]. It is based on a semantics defined by mathematical modeling of multiple-clocked flows of data and events. Relations can be defined on such data and event signals to describe arbitrary dynamical systems and then constraints may be used to develop

real time applications. There are operators to relate the clocks (according to some clock calculus) and values of the signals. SIGNAL can be also described as a synchronous data-flow language.

Although SIGNAL is not designed specifically for robotics, the characteristics of robot programming and SIGNAL's mentioned functionalities makes it possible to use it for active vision-based robotics systems [4]. Since the vision data have a synchronous and continuous nature, it can be captured in signals and then the control functions between the sensory data and control outputs can be defined. The SIGNAL-GTi extension of SIGNAL can be used for task sequencing at the discrete level.

SIGNAL-GTi enables the definition of time-intervals related to the signals and also provides methods to specify hierarchical preemtive tasks. Combining the data-flow and multitasking paradigms results in having the advantages of both the automata (determinism, task sequencing) and concurrent programming (parallelism between tasks). using these advantages a hierarchy of parallel automata can be designed. The planning level of robot control does not have a counterpart in SIGNAL but the task level can be used for this purpose.

```
process MOUSE = (integer DELTA)
  { ? event TICK, CLICK
    ! event SINGLE, DOUBLE }
  (| (| START := NOT_IN_INTERVAL {CLICK, START, RELAX}
      | (| N := COUNT_IN_INTERVAL {TICK, START, RELAX} cell event N
         | ZN := N $1
         | N ^= CLICK default TICK
         | RELAX := TICK when (ZN = (DELTA-1)) |)
  | (| DOUBLE_CLICK := ((not START) default IN_INTERVAL {CLICK,START,RELAX})
                        cell RELAX
     | SINGLE := RELAX when (not DOUBLE_CLICK)
     | DOUBLE := RELAX when DOUBLE_CLICK |)
  |)
 where event START, RELAX; integer N, ZN init 0; logical DOUBLE_CLICK
end
```

This example is taken from [8]. The output signals `SINGLE` and `DOUBLE` occurs after `DELTA` time steps each time the mouse button is pressed. If during that interval another mouse click occurs then `DOUBLE` is signaled, otherwise `SINGLE` is signaled. Each equality defines a new signal from existing ones in a number of ways like delaying the original one `(ZN := N $1)` or filtering by `when`.

2.7 Charon

Charon is a language for modular specification of interacting hybrid systems and can be also used for defining robot control strategies [1]. The building blocks of the system are agents and modes.

An agent can communicate with its environment via shared variables and also communication channels. The language supports the operations of composition of agents for concurrency, hiding of variables for information encapsulation, and instantiation of agents for reuse. Therefore complex agents can be built from other agents to define hierarchical architectures.

Each atomic agent has a mode which represents a flow of control. Modes can contain submodes and transitions between them so it is possible to connect modes to others with well-defined entry and exit points. There are some specific entry and exit points. The former is used for supporting history retention, i.e. default entry transitions are allowed to restore the local state from the most recent exit. A default exit point can be used for group transitions which apply to the all submodes to support exceptions.

	Anim.A.A.	Beh. L.	Charon	Colbert	PRS	Signal	TDL	FROB
Sync/Async.	Async.	Async.	Sync.	Sync.	Async.	Sync.	Sync.	Sync.
Language	RAP	Beh. L.	Charon	Ext.(C)	PRS	Signal	Ext.(C++)	Haskell
Style of Prog.	Spec.	Spec.	Spec.	Imp.	Spec.	Spec.	Imp.	Functional
Values	NA	D,E	C,D	NA	NA	C,D,E	NA	C,D,E
Formalism	Task-Net	AFSM	Modes (FSM)	Activities (FSA)	Task Graph	Data-flow Language	Task Trees	Streams Tasks(FSA)
Concurrency	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Scheduling	Yes	No	No	Yes	Yes	No	Yes	Yes
Verification	Unclear	No	Suitable	Unclear	No	Yes	Suitable	Suitable
Signaling	Yes	No	Yes	Yes	Yes	No	No	No

Figure 5: Language Comparison

Transitions can be labeled by guarded actions to allow discrete updates. In a discrete round only one atomic agent will be executed and the execution will continue as long as there are enabled transitions. Since a mode can contain submodes group transitions are examined only when there are no enabled transitions in the submodes.

Another update happens for the variables declared as analog. This time, it is called continuous update as these variables represent continuous flow. The evolution of analog variables can be constrainted in three ways: differential constraints, algebraic constraints, and invariants which limit the allowed durations of flows. Each agent has its own clock and it is assumed that the differences between these local cloacks are bounded. In any time round, only one agent's analog variables are updated and the state of the others are frozen since the guards and invariants of an agent can depend on these updated values.

2.8 Summary

The table shows some of the different capabilities and design choices of the languages and FROB which will be introduced in the next section. Since some of these languages are components of a larger system and dedicated to one aspect it is not possible to compare them in overall. Some of them are more general frameworks which are used also for robot programming (i.e. Signal, Charon), while some of them are used for a certain part of the system (like TDL for task scheduling, PRS for planning and execution).

The most common feature of these languages is the mechanisms for defining some components, building blocks which correspond to the notion of tasks and/or behaviors. They all support modularity through hierarchical composition of these components and instantiation. As required by the nature of robot programming, some form of concurrency is also available in all of them (i.e. multiple tasking in a single process or distributed processes). Except Behavior Language which is used for implementing subsumption architecture, they all have reactive and planning parts.

Colbert, TDL, Animate Agents Arhitecture and PRS are mostly focused on task scheduling. Colbert uses signaling to achieve this end (i.e tasks can interrupt and resume other ones by using their names). Others have special semantics to coordinate the spawning and end time of tasks. One can specify the termination criteria or the beginning time of a task depending on the lifespan of others. We assume that languages in which the tasks can read and write to ports (channels) with a specific name also have the signalling ability since it is possible to change the control flow indirectly through these ports.

Robot programming may require three kind of values to be specified and used. First one is the ability to

encode and compute continuous specifications like differential equations. These will generally depend on regularly sampled values which will probably come from the world through sensors. Another datatype is the asynchronous discrete events which will generally correspond to the chaotic nature of the environment. We denote these values by C(ontinuous), D(iscrete) and E(vents) respectively. The languages focused on the task scheduling mainly does not need to include such values.

Error handling is another common feature of the languages. Exceptions and error related signals can change the control flow of the tasks. First of the two general approaches to this problem can be best observed in RAPs. A `wait-for` statement can check an error signal just like a termination criteria and determines the continuation. Alternatively, like TDL's exceptions, the definition of a task can include an exception statement with an exception name and the related exception handler. In this approach, handling an exception can be throw up where an ancestor task will determine how to handle the excpetion.

Formal verification of the robot programs are also desirable in the robot control domain. More formal languages like Charon and Signal have an advantage for this purpose. The need for simulation, debugging, testing and formal verification is recognized by the language developers and future work generally includes plans to develop such visualization and analysis tools.

TDL and Colbert are the languages which can be described as extensions and C being their base language they have an imperative style. Other languages have their own syntax in which -with the exception of Signal, Charon- the specification of a component and its relations to the other componenets (tasks or behaviors) has a very LISP-like flavor.

3 FROB

3.1 FRP

Functional Reactive Programming (FRP) is a general framework for hybrid systems which employs behaviors and events as its basic building blocks [25]. Behaviors are continuous sequences that vary over time while events represent discrete asynchronous or synchronous event occurences. FRP's declarative style enables rapidly developing modular, reusable high-level programs. The principles underlying current FRP framework were first introduced in Fran [5], a domain-specific language for programming reactive animations. The host language for Fran was Haskell which gives FRP the flavor of declarative programming [21] [11]. FROB (for Functional Robotics) is another domain-specific language embedded in Haskell based on FRP which is designed for robot control [19] [20] [9]. FRP is now also used for vision and other control systems applications [22] [12]. Haskell, the host language of FRP, is a higher-order, polymorphic, typed, lazy and purely functional language. The lazyness and type systems of Haskell were exploited in FRP's implementation.

Both behaviors and events, which are the fundamental data types of FRP, are based on streams. A behavior of type `a` can be defined as a mapping from time and some type of input (again which can be represented by a stream) to a stream of values of type `a`. Behaviors have continuous semantics in the sense that they always have a value whenever they are sampled (i.e. an animation of a ball rotating around a point, sonar readings of a robot, mouse position, velocity of a robot during some navigation related task). In contrast events represent discrete-time event occurences which means they only have values at particular points of time (i.e. left button pressed, robot bumped to an obstacle). Since events are also represented by streams, the related event occurences are time-ordered.

```
type Behavior inp a = Stream inp -> Stream Time -> Stream a
type Event inp a    = Stream inp -> Stream Time -> Stream (Maybe a)
```

These simplified type declarations show that both behaviors and events are time dependent mappings from an input stream (i.e. an infinite list) to an output stream. The values of the event stream are either `Nothing` denoting no occurence or `Just a` which means the event occured and its value is `a`. This characteristic is captured by the Haskell `Maybe` type. We will drop the `inp` type variable from types of behaviors and events from now on for the sake of simplicity (i.e. so possible types of the example behaviors above will be : `Behavior Graphic` for animation of a ball, `Behavior Image` for sonar readings, `Behavor Point2` for mouse position, and `Behavior Real` for velocity).

There is a rich library of combinators and transformers in FROB to produce new events and behaviors from existing ones. Here are some of the most basic ones.

```
(-=>) :: Event a -> b -> Event b
```

`e -=> x` is a new event with the constant value `x`, occuring only (and always) when `e` occurs.

```
(==>) :: Event a -> (a -> b) -> Event b
```

Similar to `-=>` above but this time the function argument is applied to the original event's value to get the new value.

```
till   :: Behavior a -> Event (Behavior a) -> Behavior a
switch :: Behavior a -> Event (Behavior a) -> Behavior a
```

These switchers can be used to transform behaviors with the help of events. `` b1 `till` e-=>b2 `` is the behavior with values from `b1` until the first occurence of `e` and then behaves like `b2`. `switch` works similarly but this time at every occurence of `e` the overall behavior changes to the behavior produced by the event `e`. For instance :

```
b1 `switch` ( (lbp -=> b2) .|. (key ==> \x -> if x='A' then b3 else b4) )
```

is a new behavior starting like `b1`, switching to `b2` *everytime* when left mouse button is pressed (`lbp` is a predefined event returning `()`) or to `b3` or `b4` depending on the key pressed (`key` is also a predefined event which returns the key pressed). Note that all behaviors `b1, b2, b3`, and `b4` must have the same type to ensure the type of the resultant behavior. If `till` were used instead of `switch`, after the first occurence of `lbp` or `key`, the final behavior will remain as `b2`, `b3` or `b4` depending on the event, ignoring other event occurences after that point.

```
(.|.) :: Event a -> Event a -> Event a
mergeE :: (a -> a -> a) -> Event a -> Event a -> Event a
```

`.|.` is used for the disjunction of events as seen in the previous example. `e1 .|. e2` occurs everytime either `e1` or `e2` occurs and the value is preserved. If both events occur at the same time, the implementation favors value of the the leftmost event to be the value of the composed event. `mergeE` can be used to apply a merger function to get the value when both events occur at the same type.

```
snapshotE :: Event a -> Behavior b -> Event (a,b)
snapshotE_ :: Event a -> Behavior b -> Event b
```

`snapshotE` is used to capture the value of a behavior when an event occurs. `snapshotE_` can be used to disregard the event's value. These are useful to define switching behaviors where the switching time is determined by an event and the new behavior depends on the behaviors current value.

```
color = red `switch` (lbp `snapshotE_` color ==> changeColor)
        where
         changeColor oldColor =  if oldColor = Red then yellow
                                 else if oldcolor = Yellow then green
                                 else red
```

In this example `red`, `green` and `yellow` are predefined constant color behaviors. The value of behavior `color` at the event occurence is captured by `snapshotE_` and then compared to `Red` and `Yellow` (in uppercase) to determine the next color behavior. `Red` and `Yellow` are predefined color values composing the constant color streams `red` and `yellow`.

```
stepB :: a -> Event a -> Behavior a
stepAccumB :: a -> Event (a -> a) -> Behavior a
```

Step functions are like switchers but this time the final behavior's values are directly determined by the event. So `a stepB e1` is a behavior which starts as `a` and then switches to `e1`'s value every time `e1` occurs.

```
e          : __ __ a1 __ a2 __ __ __ __ a3 a4 __ a5 __ __ ...
a0 step e : a0 a0 a1 a1 a2 a2 a2 a2 a2 a3 a4 a4 a5 a5 a5 ...
```

In the case of `stepAccumB` the value of the event (which is a function) is applied to the current value of the behavior to determine the next values of the behavior which will be constant until event occurs again. For instance :

```
counter = 0 `stepAccumB` lbp -=> (+1)
lbp     : _ _ | _ | | _ _ _ | _ _ | _ _ ...
counter : 0 0 1 1 2 3 3 3 3 4 4 4 5 5 5 ...
```

is a counter of the left mouse buttons pressed.

```
whenE :: Behavior Bool -> Event ()
b       : F F F T T F T F T T T F F T F ....
whenE b : _ _ _ | _ _ | _ | _ _ _ _ | _ ....
```

`whenE` can be used to turn a boolean behavior to an event. The event occurs everytime the behavior changes from `False` to `True`.

```
($*)  :: Behavior (a->b) -> Behavior a -> Behavior b
lift0 :: a -> Behavior a
lift1 :: (a->b) -> Behavior a -> Behavior b
lift2 :: (a->b->c) -> Behavior a -> Behavior b -> Behavior c
lift3 :: (a->b->c->d) -> Behavior a -> Behavior b -> Behavior c -> Behavior d
....
```

The lift operators are used for transforming behaviors (in fact they can also be used for events by the help of overloading and type classes). `lift0` returns an constant behavior with the value of its argument. `lift1` maps its first function argument to its second behavior argument. `lift2` zips two behaviors by applying its function argument to the corresponding values of the behaviors. The operator `$*` can be used to define the family of lift functions by the help of lift functionss at the below level. `$*` applies the values of its first argument (a behavior of functions) to its second argument to get the final behavior. So for example :

```
lift3 f b1 b2 b3 = lift2 f b1 b2 $* b3
```

Because of the function currying of Haskell `lift2 f b2 b2` will return a behavior of type `c->d` which by the help of `$*` will be applied to `b3` of type `Behavior c` to return the final behavior of type `Behavior d`.

There are also clocked events of type `CEvent clock inp a`. The `clock` type of a clocked event determines the occurence pattern of the event. So if two events, e1 and e2 have the same clock, then they are guaranteed to be synchronous which means e1 occurs whenever e2 occurs. Therefore one can safely apply lifted functions to synchronous events since the Haskell's type system prevents events with different clocks to be mixed.

3.2 FROB

FROB is the extension of FRP for robot controlling. The Task monad of FRP is used to define tasks and sequence them for modular programming. A task combines a behavior with a terminating event.

The basic type used in FROB is `RController` which defines a behavior mapping from stimulus to a control value :

```
(Sense1 i, Sense2 i, ..., Effect1 o, Effect2 o, ...) => RController i o
```

Here the type constraints restrict the input and output of the behavior to a specific class. Each sensor input class defines some set of behaviors or events which can be used for defining the controller. Similarly for each effector class there are functions which will be used for controlling the robot.

```
position      :: Odometry i => Behavior i Point2
heading       :: Odometry i => Behavior i Angle
stuck         :: StuckDetection i  => Event i ()
robotDiameter :: RobotProperties i => Behavior i Length
```

These are some of the signals related to the input classes. If the input of the controller is constrained as having `Odometry`, the `position` and `heading` behaviors, which give the current position and heading of the robot, can be directly used for defining the controller.

```
setWheelSpeeds :: WheelControl o   => Behavior i (Speed, Speed) ->
                                      RController i o
setHeading     :: HeadingControl o => Behavior i (Speed, Speed) ->
                                      RController i o
setBuzzer      :: Buzzer o         => Event i (Frequency, DeltaTime) ->
                                      RController i o
```

For output type classes, there are functions which define a robot controller depending on behaviors or events. These functions can be viewed as the commands that robot understands. Let's see an example how these are used for defining controllers:

```
gotoPoint :: (Odometry i, WheelControl o) => Point2 -> RController i o
gotoPoint p = setWheelSpeeds (pairZ fwdSpeed rotSpeed)
              where
              fwdSpeed = ....     -- position and heading behaviors of
              rotSpeed = ....     -- Odometry can be used to define these
```

The important point is that `gotoPoint` is independent of the hardware of the robot. For using a robot with FROB one has to define only specific behaviors and functions related to input and output classes. Then robots having same sensors and actuators can use the same controllers without changing their code. `pairZ` gets two behaviors and returns behavior of pairs composed by these two behaviors.

Tasks can be used to generate complex robot behavior. A robot task is a behavior (robot controller) combined with a terminating event and a state:

```
type RobotTask s i o e = ...
```

Here, `s` denotes the type of the internal state of the robot. `i` is used for the input robot senses, `o` specifies the effectors and finally `e` shows the type of the terminating event which also determines the exit value of the task.

The Task monad can be used to sequence tasks. For example in :

```
t = do t1
       t2
```

`t` is a new task combining tasks `t1` and `t2` where the overall behavior of `t` is the combination of behaviors of `t1` and `t2`. When `t1` finishes, the behavior is switched to `t2`.

```
t = do x <- t1
       t2
```

In this type of binding the result of the task `t1` is stored in the variable `x` and can be used while defining `t2`. `return x` results in a task which exits immediately with the value `x`.

```
mkTask      :: Behavior i o -> Event i x -> RobotTask s i o x
liftT       :: Behavior i o -> RobotTask s i o x
liftT b     =  mkTask b neverE
```

The basic building block for tasks is the `mkTask` function. `liftT` can be used to lift behaviors to tasks which never terminate.

```
bmapT :: (Behavior i a -> Behavior i b) -> RobotTask s i a x ->
                                          RobotTask s i b x
fmap  :: (x -> y) -> RobotTask s i b x ->
                     RobotTask s i b y
```

bmapT and fmap can be used to change the associated behavior and event value of a task respectively.

```
  snapshotT    :: Behavior i a -> RobotTask s i o x ->
                                  RobotTask s i o (x,a)
  snapshotNowT :: Behavior i b -> RobotTask s i a b
```

snapshotT runs a behavior parallel to the task and adds the value of the behavior to task's exit value. snapshotNowT results in a task which immediately returns with the current behavior value.

```
tillT    :: RobotTask s i b x -> (Behavior i b -> Event i x) ->
            RobotTask s i b x

tillT_   :: RobotTask s i b x -> Event i x -> RobotTask s i b x
```

tillT_ is used for adding a termination event to an existing task. The new task terminates either the old or the new termination event occurs. It is generally used for adding termination criteria to non-terminating tasks like the ones generated by liftT. By using tillT, the behavior of the task can be used to define the new terminating event.

```
timeLimitT  :: DeltaTime -> RobotTask s i a e -> RobotTask s i a (Maybe e)
timeLimitT_ :: DeltaTime -> RobotTask s i a e -> RobotTask s i a ()
```

These are used for limiting the time a task can run. timeLimitT returns the normal task exit value if it terminates before the time given. Otherwise the new task returns Nothing. timeLimitT_ returns () for both cases.

```
  (|.|) :: RobotTask s i o e -> RobotTask s i o e -> RobotTask s i o e
```

To run tasks in parallel |.| is used. In current implementation of FROB the two parallel tasks can try to control the same effectors and the type system does not warn the programmer. In such a case, one of the tasks controller will be ignored.

```
  getState  :: RobotTask s i o x
  setState  :: (s -> s) -> RobotTask s i o x
  sendT     :: Message -> RobotTask s i o ()
```

If the task has an internal state, getState can be used to access it and setState can be used to modify it. Here the output task of getState is a task which immediately exits with a value. sendT is another instantaneous task which sends a message.

4 Comparative Analysis

In this section we will try to find out the strengths and weaknesses of FRP and FROB. For this purpose, first we will look at how some common architectures can be expressed in FROB's framework, how much flexible FROB is to different design choices (i.e. behavior coordination strategies, planner and reactor integrations) and then we will rewrite examples from some of the languages given in the previous section.

4.1 Finite State Automata

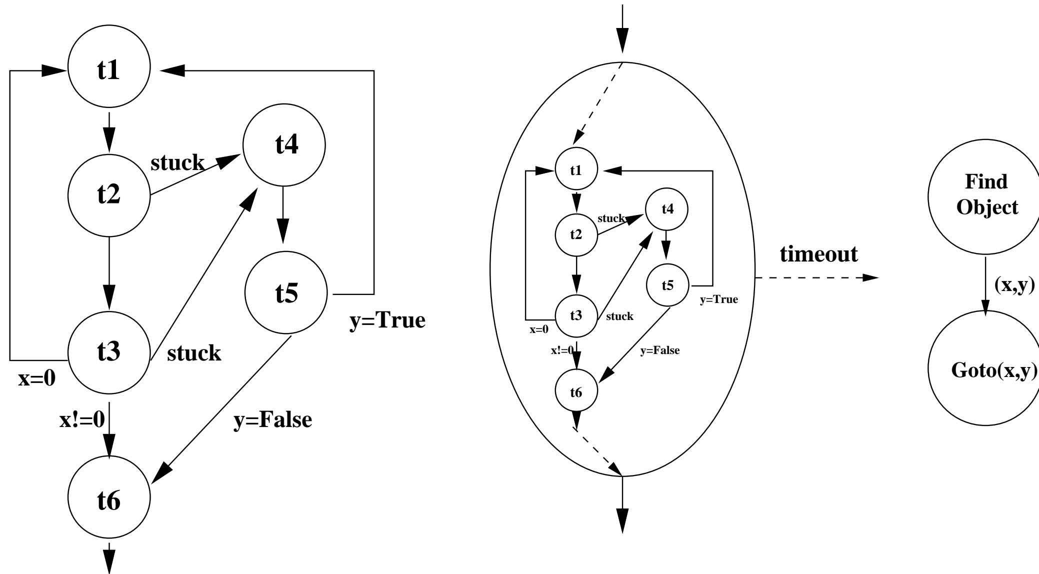

Figure 6: FSA representation of tasks

Finite state automata can be used to represent the sequences and aggregations of behaviors which define a complex task. They are not powerful enough to show how a behavior is encoded but a FSA diagram intuitively represents the transitions between the behaviors. In robot control language domain, a node of a FSA denotes the active behavior and a transition to a new node means switching to a new behavior.

In FROB, the task abstraction pairs a behavior with a terminating event. The task monad defines how to bind two tasks. By using the monadic `do` notation, it is pretty straightforward to express the FSA of figure 6:

```
t = do t1
       t2 `tillT_` stuck -=> handleStuck
       x <- t3 `tillT_` stuck -=> handleStuck
       if x=0 then t
              else t6
    where
       handleStuck = do t4
                        y <- t5
                        if y then t
                             else t6
```

The result of the previous task (i.e. the terminating event value) can be used to choose from different continuations. The loops can be expressed by recursive definitions or recursive task functions like `repeatT` or `foreverT`.

The nodes can be both primitive actions or complex behaviors. So we can use `t` as a simple task node where its final subtask `t6`'s value and termination time will determine its own. The transitions between the subnodes will be hidden to the new FSA but it is possible to modify the behavior or the result of `t` by `bmapT` and `fmap` respectively. One can also add new termination criteria to the composite node and this will apply to all the subtasks. For example if a time limit criteria were added to `t`, the `timeout` transition will be taken

when the time exceeds a specified time amountwithout considering which subtask was active at that point. In fact, we can use this property of FROB to define `t` alternatively as:

```
 t = do t1
        x <- do {t2; t3} ‘tillT_‘ stuck -=> handleStuck
        if x=0 then t
               else t6
....
```

Although it is not common in FSA diagrams, in FROB, the result of a task can be also used to instantiate the next task as in figure 6:

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

The basic weakness of FSA diagrams is the difficulty of representing concurrency and interactions between concurrent tasks.

4.2 Robot Schema (RS)

Lyons and Arbib define Robot Schemas as a special model of computation appropriate for sensory-based robot controlling [18]. By using these schemas both the plan and the environment can be represented. A basic schema consists of a schema name, input port list, output port list, a variable list for the internal variables, and finally a behavior which the schema represents.

Lyons later define robot action plans as networks of concurrent processes [17]. Process combination operators are used for building networks of processes. In FROB these operators can be expressed by using the task monad:

Sequential	P;Q	do {P;Q}
Concurrent	$P \mid Q$	$P \mid\mid\mid Q$
Conditional	$P < v > @ : Q_v$	do { $v \leftarrow P$; Q v }
Disabling	P#Q	$P \mid . \mid Q$

Here the operator `|||` is different than `|.|` the parallel operator defined earlier in the sense that the parallel tasks can observe each others behaviors:

```
  (|||) :: (Behavior i o -> RobotTask s i o e) ->
           (Behavior i o -> RobotTask s i o e) ->
   RobotTask s i o e
```

So instead of connecting input and output ports of concurrent behaviors one should pass the behaviors between the tasks. This is still somehow limiting in current FROB since the behavior encoded in a robot task should be of type `RController` but the processes can share any information via their ports. For the examples where these ports are used for connecting sensors and effectors, these are handled by FROB's own mechanisms for getting sensory data and sending commands to the robot. The other two composition operators can be defined using the above ones.

The mixed-batch knitting problem [17] which Lyons used to show how to compose processes can be solved in FROB as :

```
part1 = do (p,t) <- locate tray
           placeTray p t
           (p,m) <- locate m1 |.| locate m2
           placeMotor p m

part2 = do (p,s) <- locate sc |.| locate sds
           case objType s of
            sds -> placeSDS p s
            sc  -> do placeSC p s
                      (p,s) <- locate s1 |.| locate s2
                      placeSwitch p s

plan = do part1
          part2
```

Lyons also gives an example how the environment can be modelled by using these process composition operators. FRP's behaviors, events and combinators are very useful for this purpose. Haskell's abstraction mechanisms and the functional programming style also makes these kind of environment modelling relatively easier and more intuitive in FROB. For example the cars in the traffic world of [17] can be defined as:

```
type Car = Behavior Point2
type Velocity = Behavior Real
datatype CarType = Avoid | Hit | Ignore
car :: CarType -> Velocity -> Car
car type vel = case type of
                Avoid  -> ...
                Hit    -> ...
                Ignore -> ...
chooseVel :: Time -> Velocity

cars :: Behavior [Car]
cars = nilZ `switch` (key `snapshot` timeB ==>
                            (\(ch,t) = let vel = chooseVel t
                                               c = case ch of
                                                   'a' -> car Avoid vel
                                                   'h' -> car Hit vel
                                                   'i' -> car Ignore vel
                                           in
                                               consZ cars c)) .|.
                     lbp -=> consZ cars (car Hit Fast) .|.
                     rbp -=> consZ cars (car Avoid Slow)
```

Here the function `chooseVel` returns a velocity behavior depending on the time. `cars` start as an empty list and then each time a key, left mouse button or right mouse button pressed, a car is added to the list.

4.3 Subsumption Architectures

The coordination process in subsumption architecture results from two basic mechanisms, namely inhibition and suppression. Inhibition means a signal is prohibited reaching the actuators and supression means the signal is replaced with another suppressing message.

We can have the similar effect in FROB by adding priorities to behaviors and using them for coordination of the behaviors.

```
type RealB = Behavior RealVal
type Stimulus a = Stimulus {value = Behavior a, priority = RealB)

priorityB :: Stimulus a -> RealB
priorityB = lift1 priority
valueB :: Stimulus a -> Behavior a
valueB = lift1 value
withPriorityB :: Behavior a -> RealB -> StimulusB a
withPriorityB b p = Stimulus {value=b, priority=p}

subsumes :: StimulusB a -> StimulusB a -> StimulusB a
subsumes x y = ifZ (priorityB x >=* priorityB y) x y
```

The basic subsumption combinator is `subsumes` where the behavior with the higher priority is returned as the function value.

Let's give an example to show how these can be used. We want to define a robot task where the robot will wander randomly while avoiding objects, pick up objects when it sees one and travels to home once all pickups are complete.

```
homing :: WheelControl o => BoolB -> Stimulus (RController i o)
homing r allDone = travelTo home `withPriorityB` ifB allDone 2 0

pickup :: (Sonars i, WheelControl o) => [Point2] ->
                                        Behavior (Stimulus WheelControl, Bool)
pickup [] = pairZ (noCommandT `withPriorityB` 0) trueZ
pickup (p::ps) = ....

wandering :: WheelControl o => Stimulus (RController i o)
wandering = randomWalk `withPriorityB` 1

avoiding :: (Sonars i, WheelControl o) => Stimulus (RController i o)
avoiding = ....

system :: (Sonars i, WheelControl o) => RController i o
system = valueB $    -- strips off the priority from the output
           let p = pickup [point2XY 200 200, point2XY 100 100]
               pickupB = fstZ p        -- unpackage the pickup behavior
               pickupDone = sndZ p     -- and the "done" flag
           in
               homing pickupDone `subsumes`
               pickupB `subsumes`
               wandering r `subsumes`
               avoiding r
```

The homing behavior takes a boolean behavior to check if all the pickups are done. Until that is true the priority is 0 (i.e. the behavior is not active), after it becomes true the priority is 2. The wandering behavior

has a constant priority of 1. So once homing is started it is inhibited. Pickup behavior's priority is 0 until a goal is seen. Once a goal is seen it raises to 5 until the goal is visited. The priority of avoiding is determined by the sonar readings. The sonar readings are combined into a force that pushes robot. So depending on the closeness of the obstacles avoiding can supress all other three navigation tasks but once the obstacle is avoided, the priority of avoiding will drop and the previous taks can continue whatever it was doing.

Pickup is initialized with a set of points representing the goals to visit. Its result is attached with a boolean behavior to specify whether all the goals are visited. This behavior is passed to homing to make it determine when to start going to home position. Note that `system` has constraints Sonars i and WheelControl o since the two of its building blocks (pickup and avoiding) are using sonars and all four of them are robot controllers that send commands to the wheel of the robot.

4.4 Assembling Behaviors

There are many behavior coordination mechanism. Since FROB is embedded in a functional language, one can easily develop the abstractions to define the coordination mechanism which is more appropriate to the problem domain.

One coordination method is making behaviors cast votes for different actions. The action receiving most votes is chosen as the control behavior. For instance:

```
type Vote = FRPReal
datatype Plan = Avoid | Goto_Goal

choose :: [(Plan,Vote)] -> RController i o
choose votes = let add = foldl 0 +
                   avoidV = add (map (\(p,v)->if p=Avoid then v else 0) votes)
                   goalV  = add (map (\(p,v)->if p=Goto_Goal then v else 0) votes)
               in
                 if avoidV>goalV then avoid
                                 else goto_Goal
```

Another competitive approach is to couple behaviors with situations and choose the highest priority behavior whose constraint become true. For example, a control strategy for the goalkeeper of a robot soccer team can compose of kicking the ball when it is too close, retreating to goal when necessary and blocking the goal by moving to between the goal and the ball.

```
type SC i o = (Behavior i Bool, RController i o)

.+. :: SC i o -> SC i o -> SC i o
(r1, c1) .+. (r2, c2) = (r1 || r2 , ifB r1 c1 c2)

goalkeeper :: (Vision i, Odometry i, WheelControl o) => SC i o
goalkeeper = stayNear .+. kickIt .+. blockIt
             where
              stayNear = (distance position goal > maxGoal,
                          travelTo goalCenter)
              kickIt   = (distance position ballPosition < maxK &&
                          abs (vector2angle (ballPosition .-. position)) < pi/2,
                          travelTo ball)
              blockIt = (trueZ, blocker)
```

```
                blocker = let v1  = ballPosition .-. position
                              v2  = ballPosition .-. goalCenter
                              vec = normV v1 - normV v2
                          in pushRobot vec
```

Here the highest priority is stayNear's. So if robot is away from the goal by maxGoal it returns to the goalCenter immediately. If this predicate is not true then the closeness of the ball is checked. If it is near then maxK the robot goes to ball to kick it. If both of these constraints are false the robot always tries to position between the ball and the goal.

Most of the cooperative coordination methods based on behavioral fusion with vector summation. FRP modules VectorSpace and Geometry can be used for this purpose.

4.5 Colbert examples

```
iterateT 1 t = do t
iterateT n t = do { t; iterateT (n-1) t }

turn angle = do initAngle <- snapshotNowT heading
                mkTask $ setHeading (lift0 (initAngle + angle, 0.5))
                         whenE (heading ==@ lift0 (initAngle + angle))

move dist vel = let dist2 = f dist in -- convert dist from meters to a vector
                do initPos <- snapshotNowT position
                   mkTask $ setWheelSpeeds (lift0 (vel,0))
                            whenE (initPos .-. position >@ dist2)

patrol1 = do move 1000 20
             turn 180
     move 1000 20


patrol n = iterateT n patrol1
```

To express the Colbert example, where robot patrols between two points for a specified number of times, in FROB we first define a function to iterate a task for a given number of times. For defining **turn** first the heading angle is stored at the task initiation. **setHeading** is used to control the robot's heading by giving a desired angle and a turning rate. The robot turns slowly until the desired heading is achieved. **move** is defined similarly.

```
approach = do x <- timeLimitT 30 $ snapshotT_ frontSonar $
                     (foreverT patrol1) 'tillT_' whenE (frontSonar <@ 2000)
              case x of
                Nothing -> error "failed due to timeout"
                Just d  -> move (d-200) 20
           'tillT_' stuck -=> error "failed due to an obstacle"
```

In this example the robot patrols for 30 seconds or until an object is detected in less than 2 meters. If an object is detected, the distance to object is returned by using **snapshotT_** and the robot moves to within

20 cm. of the object. Note that how the non-terminating patrol task is constrained by a terminating event and a time limit. Similarly the overall approach task is terminated if the robot stucks at some point.

In the next Colbert example the robot patrols n times but patroling may be interrupted by other actions. There is no interruption mechanism in the current FROB but sending and receiving messages can be used to implements kind of task interaction. If the input stream is an instance of Messaged type class then we can assume that there is a `signals` events stream for this purpose.

The Colbert example is designed such that if the robot is interrupted while there are n more patrols to finish including the current one then when it is resumed it makes n more patrols. So first we change our patrol definition to get our hands on this value. We will use the internal state of a task to accomplish this.

```
patrol n = do res <- loop
              if res=Success then
                return ()
              else
                do stopT ‘tillT_‘ (whenE signals =@ lift0 (Resume "patrol"))
                   m <- getState
                   patrol m+1

           where loop = do setState (+1)
                           patrol1
                           m <- getState
                           if m=n then return Success
                                  else loop
                        ‘tillT_‘ (whenE signals =@ lift0 (Interrupt "patrol")
                                  -=> Failure)
```

4.6 RAP system

First we will look at the RAP example where a tracking and approaching task was defined to illustrate the capabilities of different task continuations, control flow between tasks, lifespan of tasks depending on others.

```
approach = do camera ON
              res <- moveTo target |.| track target
                 ‘tillT_‘ lost-target -=> LostTarget
                         ‘tillT_‘ camera-problem -=> Fail
                         ‘tillT_‘ stuck -=> Stuck
              camera OFF
              case res of
                LostTarget -> ...
                Stuck      -> ...
                Fail       -> ...
                Success    -> ...
```

In this example `moveTo` and `track` are defined as parallel tasks. Then terminating events are added to this composite task. The camera is shutdown regardless of how this task ends and then different continuations can be taken depending on the result of these two parallel tasks. Firby gives this example to show that only success and failure as a task result is not enough for complex behaviors. In FROB the task results can be

any value and the existent tasks' results can be modified in a number of ways. The task monad enables to sequence tasks in a flexible way and also to choose from different continuations depending on the results of the previous tasks.

The second example from RAP system shows how to handle spawning tasks in subtasks. Unfortunately in current FROB this kind of spawning tasks in a subtask and then terminating them in another subtask is not possible. The `monitorHand` task must be active during all the `move-object` task:

```
move-object container = do pickup-object
                           goto container
                           drop-off-object)
                        |.|
                        monitorHand
```

This results in using the computational power unnecessarily in some part of the subtasks but if the added task is a monitor like in this example, Haskell's laziness prevents computing anything that will not be used. The Task monad can be extended having an environment which will be passed to the next task. Then it may be possible to carry this monitoring task in the environment. In that case the code will possibly look like:

```
move-object container = do pickup-object
                           goto container
                           drop-off-object

pickup-object = do start-pickup
   addEnv monitorHand
                   grasp
                   finish-pickup

drop-off-object = do start-drop-off
                     ungrasp
                     if (inEnv monitorHand) then delEnv monitorHand
                                           else return ()
                     finish-drop-off
```

4.7 TDL

The TDL example given at the previous section defines a task where a mail delivary robot navigates to a room and tries to deliver the mail. This task can be defined in FROB as:

```
deliverMail room = do (x,y) <- getRoomCoordinates room
                      navigateToLocn (x,y)
                      timeLimitT 30 centerOnDoor
                      monitorPickup |.| speak "Xavier here with your mail"

centerOnDoor = do whichSide <- lookForDoor
                  if (whichSide != 0 ) then
                     if (whichSide<0) then
                        move -10
                     else
                        move 10
```

```
                centerOnDoor
             else stop
```

It is not possible in FROB to expand future task trees while executing previous ones as in TDL since spawning means function call in FROB but much of the functionality in this TDL example can be captured. Expanding future task trees does not gain too much computation time but can be useful for building complex plans. TDL's `sequential execution` is the default for FROB in task monad and while TDL spawns tasks in parallel by default, in FROB `|.|` combinator is used as seen in the example where speaking activity and `monitorPickup` runs in parallel.

4.8 Signal

```
mouse = loopOut lbp
        where
        loopOut (Nothing 'consS' es) = Nothing 'consS' loopOut es
        loopOut (Just () 'consS' es) = Nothing 'consS' loopIn delta False es
        loopIn 0 clicked (_ 'consS' es) = Just y 'consS' loopOut es
                                          where
                                            y = if clicked then Double
                                                           else Single
        loopIn n x (Nothing 'consS' es) = Nothing 'consS' loopIn (n-1) x es
        loopIn n x (Just () 'consS' es) = Nothing 'consS' loopIn (n-1) True es
        delta = 10
```

In this SIGNAL example rather than following the defining signals in terms of existing signals approach we found it more convenient to get the desired effect by going into the signals, i.e. checking the stream values. The lazy evaluation feature of Haskell enables us to deal with infinite datatypes like streams.

In this example, `lbp` is the predefined event stream related to left mouse button and `consS` is used for constructing streams by giving the head and tail of the stream. The definition depends on the values of the `lbp` event stream. When an click is detected which means a `Just ()` value in the event stream, the `loopIn` function is called. The first input value denotes how many more clock cycles are there to exit from the interval. It is reset as 10 and decreased by each `loopIn` call. The second boolean value denotes whether another mouse click occured inside the interval. It starts as `False` and set to `True` if an click is observed. The next input value is the rest of the stream, since the head is processed in this call. When the first input value of `loopIn` hits 0, meaning the interval has ended, the boolean value is checked and a `Double` or `Single` value is fired according to this value. The `loopOut` function is called with the rest of the stream which again starts to wait for a click to initiate `loopIn`. This example shows how the Haskell's functional style can be incorporated into what we want to define in FRP as pattern matching and recursion helped a lot to define `mouse` in this more intuitive way.

5 Conclusions

After the review of the robot control languages and comparing examples in FROB and in these, the strengths of FROB emerge as:

- FROB is an embedded language in Haskell. So all the power of the functional language Haskell can be used in robot programming. Since the functional paradigm depends on seperating the what-to part

from how-to, the FROB definitions are more intuitive and formal definitions of tasks can be almost directly coded in FROB. This makes FROB a good choice for rapid prototyping which is important in the robot domain because of its experimental program development nature.

- FROB does not have a fixed approach to robot programmimg. So one can use whatever architecture best suits her needs. For example using both the FSM depended task definitions or subsumption architecture is possible and easy in FROB. Since developing abstractions is easier in Haskell, the programmer has more flexibility, like using the task arbitration method which suits best the goal of the task.
- Similarly most of the languages reviewed gives one a set of possible things which can be done and methods to do these. For the end-user of these languages it is impossible to extend or improve these languages. Since FROB is basically just Haskell code, one has more options and feels flexible in FROB from this point of view.
- It is easier to adapt FROB to different type of robots and again one does not have to be the language's developer to do this. Also since the low level details are hidden under abstractions, program development can be done independently from the specific hardware.
- FROB is based on FRP and FRP has a rich set of operators to define events and behaviors. This can be useful at behavior-based robotics since defining more complex behaviors from existing ones is more straight forward and can be done more intuitively.
- The basic building blocks of robotic systems and general control strategies of robot programming are available in FROB and these can be used in a modular and reusable way. As the examples show the monadic definition of tasks gives FROB enough strentgh for scheduling tasks, constructing plans and building more complex tasks from simpler ones.
- The mathematical nature of FROB and Haskell makes it easier to reason about programs and program tranformations.
- Other systems like FVision which also depend on FRP can be easily integrated into the FROB framework.

The weaknesses of FROB are :

- Using a functional language also has its drawbacks like garbage collection can decrease the performance in real-time applications. In fact there is on going work to have FRP-like constructs in C++ and transforming FRP programs into languages like C++ or Java. So the ideal solution can be to develop programs in FROB by all the advantages of functional programming and then directly encoding or automatically translating these into a more low-level language like C or C++.
- The concurrency in FROB is based on having two or more tasks which control the different effectors run simultaneously. The type system of Haskell does not allow at this point to detect two conccurrent tasks which try to use the same effector. This kind of concurrency also is not powerful enough to model interactions between tasks. Interruptions and sending messages between tasks are generally indirectly expressed in the current FROB or impossible for some cases.
- There is not any seperate exception abstraction in FROB but the termination criteria of a task can be modified by adding events. This is generally adequate for most of the cases but the continuation should be coded seperately and this can be in conflict with the modularity.

- Similarly sometimes having a global database shared by tasks or ability to refer tasks with its names are useful in other languages. FROB's functional style results in slightly more complicated code.

References

[1] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.

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

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

[4] H. Marchand E. Marchand, E. Rutten and F.Chaumette. Specifying and verifying active vision-based robotic systems with the signal environment. *The International Journal of Robotics Research*, 17(4), April 1998.

[5] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX, October 1997.

[6] R. J. Firby. Task networks for controlling continuous processes. In *Proc. of the Second Int. Conf. on AI Planning Systems*, Chicago, IL, 1994.

[7] R. J. Firby. Modularity issues in reactive planning. In *Proc. of the Third Int. Conf. on AI Planning Systems*, 1996.

[8] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real time applications with signal. In *Proceedings of the IEEE*, volume 79, 1991.

[9] 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.

[10] Vincent Hayward and & Richard 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.

[11] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[12] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

[13] 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.

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

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

[16] Kurt Konolige, Karen Myers, and Enrique Ruspini. The saphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9, 1997.

[17] D.M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(3), June 1993.

[18] D.M. Lyons and M.A. Arbib. A formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation*, 5(3), June 1989.

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

[20] 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.

[21] S. Peyton Jones (ed.). Haskell 98: A non-strict, purely functional language. Technical Report RR-1106, Yale University, February 1999.

[22] 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.

[23] Vitor M. Santos, Jose P. Castro, and M. Isabel Ribeiro. A nested-loop architecture for mobile robot navigation. *The International Journal of Robotics Research*, 19(12), December 2000.

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

[25] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles, 2000.