

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Functional Reactive Programming as a Framework for Hybrid Systems and Robot Programming

by

Izzet Pembeci

A dissertation submitted to The Johns Hopkins University in conformity
with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

August, 2003

© Izzet Pembeci 2003

All rights reserved

UMI Number: 3107560



UMI Microform 3107560

Copyright 2004 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

In this thesis, we explore Functional Reactive Programming (FRP) as a framework for robot programming and specifying hybrid systems. We present a dialect of the FRP-based domain-specific language Functional Robotics (Frob) in the latest FRP implementation. Frob enables development of modular, reusable, and flexible robot software in a declarative style. To see how Frob fits in the big picture of robot programming and test its capabilities, we present a comparative review of the recent robot languages and Frob. Additionally, we present a visual tracking-based robot navigation system, a complex real-world application, developed in Frob and show how different domains like robot programming, vision, and user interfaces can be seamlessly integrated into the FRP framework. From a more formal point of view, we also look at the relationship between FRP and hybrid systems. We show how any given formal hybrid system specification can be expressed in Frob. A subset of FRP is defined as a grammar and a syntactic translation procedure is given to convert FRP programs in this grammar to a faithful hybrid system specification.

Advisor: Gregory D. Hager

Readers: Scott Smith, John Peterson

Acknowledgements

Thanks to ...

... Greg Hager for his guidance through out this research. Each time I heard horrible PhD advisor stories, I realized how I was lucky to have him as an advisor. I am also grateful to him for his help with my job search.

... Scott Smith for being my co-advisor and a great guy.

... John Peterson for his support, valuable advice and not minding traveling to Baltimore.

... Darius Burschka, Xiangtian Dai, Linda Rorke, Bridget Nguyen-Ngoc, Aydın Selçuk, Amitabh Bagchi, and Amitabh Chaudhary for helping to solve my various kinds of problems rised during my PhD days and their friendship.

... Ali & Özgül for all the fun we have together, all the problems of *gurbet* shared. If they had not been here when I come, I would probably give up in the first years. Thanks to Tuğba & Barış and all *eskidostlar* members to allow me to take a break with the endless *geyik*.

... my family for making this possible. Thanks to Dad for keeping me on track with his PhD warnings, to Mom for bearing my being too far away for too long time, to my *biricik* sister Nurcan for the wonderful time we spent together one semester.

Finally, thanks to my wife, Barış, for sharing the hardships and stress of this part of our long journey together. I wish, I were a more organized, hardworking student and did not have to leave her alone so many nights and weekends.

To whom it may concern.

Contents

Abstract	ii
Acknowledgements	iii
List of Figures	vii
1 Introduction	1
1.1 Background	2
1.1.1 Hybrid Systems	2
1.1.2 Functional Reactive Programming	4
1.2 Goals and Motivation	6
1.2.1 A DSL for Robot Programming	6
1.2.2 Hybrid Systems and FRP	8
1.3 Approach	9
2 Functional Reactive Programming and Frob	11
2.1 AFRP	11
2.2 Functional Robotics aka Frob	17
2.2.1 Tasks	19
3 A Comparative Review of Robot Programming Languages	22
3.1 Robot Programming	22
3.2 A Review of Programming Languages	24
3.2.1 Colbert	24
3.2.2 Animate Agent Architecture	31
3.2.3 Behavior Language	37
3.2.4 PRS	38
3.2.5 TDL	40
3.2.6 SIGNAL	42
3.2.7 Charon	44
3.3 Frob Flexibility	46
3.3.1 Finite State Automata	46

3.3.2	Subsumption Architecture	51
3.3.3	Assembling Behaviors	54
3.4	Discussion	56
4	A Tracking-based Robot Navigation System	60
4.1	Implementation	63
5	Hybrid systems and FRP	73
5.1	A formalism for hybrid systems	75
5.2	Expressing Hybrid Systems in FRP	78
5.3	FRP programs as a Hybrid System	84
5.4	HS-FRP and RT-FRP	90
5.5	Translation	92
5.5.1	Bug Algorithm as a Hybrid System	103
6	Conclusions	109
A	A Translator from FRP to Hybrid System Specifications	112
	Bibliography	129
	Vita	133

List of Figures

2.1	Visualizing the signal function syntactic sugar	16
3.1	Colbert Patrol and Approach Activities	27
3.2	Colbert Patrol Activity with Interruption	28
3.3	Colbert Patrol Activity Defined in Frob	29
3.4	Colbert Approach Activity Defined in Frob	30
3.5	Frob Patrol Code with Interruption	32
3.6	RAP task-net example	34
3.7	RAP Example	35
3.8	RAP Approach Example in Frob	36
3.9	A Task Tree for Mail Delivery	42
3.10	TDL Example in Frob	43
3.11	Signal Example for Mouse Clicks	44
3.12	Signal Example in Frob	45
3.13	Expressing FSA's in Frob	47
3.14	A task example	48
3.15	Prioritized Behaviors for Behavior Merging	55
3.16	Language Comparison	56
4.1	Tracker Components	65
4.2	Tracker Abstraction in Frob as a Signal Function	65
4.3	Creating Trackers with Tasks	66
4.4	Main task for Tracking-based Navigation System	69
4.5	Mode Switchings of the Navigation System based on Events	70
4.6	User-interface while robot is navigating towards the tracked object	70
5.1	Hybrid system for the temperature control system	76
5.2	An Abstraction for Hybrid Systems	80
5.3	The Temperature Control Example	81
5.4	Hybrid system for the movement of a billiard ball	82
5.5	Billiard Ball Hybrid System in FRP	83

5.6	Grammar for HS-FRP	85
5.7	Bug Navigation Algorithm	104
5.8	The hybrid system for the bug algorithm	107

Chapter 1

Introduction

This thesis is about hybrid systems, a hybrid system framework called Functional Reactive Programming (FRP) and a domain specific language for robot programming developed using FRP. The main problems dealt with are the formal relationship between FRP and hybrid systems and how FRP can be employed for robot programming.

This thesis is organized as follows: The first chapter gives some background information about hybrid systems, FRP, and describes the goals of this thesis and the motivation behind them. The second chapter presents FRP and the domain specific language for robot programming developed in FRP, namely Functional Robotics (Frob). Chapter 3 is a comparative review of recent robot programming languages and analyses how Frob fits into the picture. Chapter 4 describes a tracking-based robot navigation system developed in Frob to show how Frob can be used to develop complex real world applications. Chapter 5 is related to the theoretical dimension of our thesis. In this chapter we first describe how an FRP program can be written which will be equivalent of a formal hybrid system definition. Then we specify a subset of FRP by a grammar where FRP programs that can be defined inside this grammar can be automatically transformed into a formal hybrid system definition. We present in detail how such a transformation can be performed. Finally, Chapter 6 is the conclusions and discusses future research possibilities based on work done

in this thesis.

1.1 Background

1.1.1 Hybrid Systems

A hybrid system is a reactive system consisting of both discrete and continuous components. Such systems typically consist of a digital program (computers) interacting with an analog environment (the physical world). They have found use in application domains such as air-traffic control systems, avionics, automated transportation, communication networks, embedded automotive control, process control, manufacturing systems and robotics. Recently, it has been shown that hybrid modeling can even be appropriate for biomolecular networks and can be used for simulation of chemical reactions related to them.

The simplest example of a hybrid system may be a temperature controller which regulates the temperature of a room by commanding a heater. When the temperature decreases below a specified value, the heater will be turned on and when it passes another specified value the controller will turn the heater off. This way, the room temperature will be maintained between two boundaries. This classical example also illustrates the two phases of a hybrid system. In the continuous phase the system evolves by time according to some rules that can be principally captured by continuous differential equations (i.e. temperature increases or decreases). In the discrete phase, which is assumed to take zero time, system goes into a new mode with new rules (i.e the heater is turned on and temperature begins to increase).

The wide applicability of hybrid systems to real world application domains motivated the formal analysis and design of hybrid systems. Such applications are generally safety critical, and thus formal analysis becomes a requirement to guarantee safe operation. Formal analysis in this case focuses on verifying whether a hybrid system satisfies a desired specification, like checking if an

unsafe region of the state space is reachable or not. Due to the complexity of large-scale hybrid systems, computational or algorithmic approaches to the verification problem are popular and desirable. [1], [29], [31], [4], [32] are some of the examples of the significant research done on the formal specification and algorithmic analysis of hybrid systems.

The modeling of hybrid systems is inspired by phase transition systems [30], timed safety automata [2], timed systems [32], and temporal logic [35]. They are generally modeled as a finite state automaton equipped with a set of variables. The transition between the nodes, i.e. system modes or states, of the automaton are labeled by guards and each location has an associated evolution law on the variable set. In the continuous phase of the system, the values of the variables are determined by the specific evolution laws of the current state. A transition is made to a new node when an associated guard becomes true.

The popularity of hybrid systems also resulted in attempts to find suitable abstractions and composition methods for specifying complex hybrid systems ([5], [45]). The most common feature of these attempts is hierarchical decomposition of the system since this makes the formal analysis easier and appropriate for algorithmic approaches. As in other domains of computer science it is also easier to design and implement hybrid systems if they can be specified consisting of different subsystems and hierarchical layers. So the main questions to be answered for better design of hybrid systems are what methods we can use to compose the subsystems, which abstractions can enhance modularity of the specification and what kind of relations can be between different layers of the hierarchy.

Despite the attention on formal analysis and design of hybrid systems, there are still a few general purpose program development environments committed to specification of hybrid systems and may support formal analysis. Charon [3] is an example of such a modeling language which allows modular specification of hybrid systems. One can describe the discrete and continuous behaviors of system's agents and the system architecture hierarchically and in a modular

way. Charon also has a simulation environment and externally defined Java functions can be used in behavior specifications.

Another programming language for descriptions of hybrid systems is Shift [12]. Shift's hybrid automata systems are dynamic in the sense that components can be created, interconnected and destroyed as the system evolves and the interaction network itself may also evolve. Shift programs can be compiled into C or an executable program for simulating the design specified.

HyCharts [18] takes a different approach and focuses on graphical specifications of hybrid systems. A modular vision model for specifying hybrid systems is important since specifications of hybrid systems will be generally discussed between engineers from different disciplines. Combining operators on hierarchic graphs and statechart-like formalisms, HyCharts enables specification of both the system architecture and component behavior graphically. There is a denotational semantics associated with each of these graphical representations. No tool support is yet available for HyCharts.

1.1.2 Functional Reactive Programming

Functional Reactive Programming (FRP) is a general hybrid systems framework for specifying programs with continuous and discrete components. FRP provides a rich set of abstractions and combinators to define complex behaviors of system agents. Another key element of FRP is the switching constructs which are used to define the evolution of the system.

FRP is based on the general notion of a signal: a value that varies over time. There are two types of signal in FRP: continuous signals that are defined at all times, and events, which are discrete signals that carry values only at specific time instances. The continuous signals represent both the environment and how the system responds to the environment. Events can be used to model discrete changes in computation. An FRP program is then about describing a mapping, in FRP jargon a signal function, from a specific stimulus to a response in a reactive manner. The mechanisms that control the time and

data flow are hidden from the FRP programmer so the program will be all about how to describe this mapping and hence will be more close to a formal specification. What kind of information may constitute the input (stimulus) and output (response) signals of the system depends on the problem domain:

Domain	Input	Output
Robotics	Sensor Information	Robot Commands
Vision	Image Stream	Graphics
GUI	User Input	Widget Displaying and State

FRP, as well as being a general framework, provides also a core for the construction and integration of specialized languages for a particular domain. In fact, the basic ideas of FRP came to life while developing a domain specific language (DSL) for reactive animation [14]. Other examples of DSLs embedded in FRP are Frob (robotics [20], [37]), FVision (visual tracking [42]), and Fruit (user interfaces [10]).

“Functional” of FRP comes from the fact that the host language of FRP is the functional language Haskell [41]. The declarative style and strong type system of Haskell supports rapid development of modular, reusable, and high-level programs. It is also easier to tune FRP to different domains by adding appropriate abstractions at the top. The performance problems of living in a functional language are avoided by exporting low-level time consuming functions (i.e. vision algorithms) from languages like C or C++ and using FRP mainly for specification of the hybrid system.

[11] describes how the declarative nature of FRP and basic FRP concepts can be captured in C++. Similarly, [9] presents Frappé, an implementation of FRP in the Java programming language. How FRP can be extended to encompass parallel systems is examined in [40]. [34] discusses another possible variation of FRP: how FRP ideas can be adapted for a non-causal hybrid modelling language like Modelica and still maintain the capability of describing highly dynamic systems. [33] gives a traffic surveillance system that utilizes visual tracking written in the latest implementation of FRP. It also presents new FRP abstractions like dynamic collections, continuation-based switching,

and time embedding which will support defining structurally dynamic systems.

The formal semantics of FRP and how it relates to an implementation based on signals that represent continuous behaviors were first explored in [47]. As an important result, it gives certain constraints to be observed on behaviors in order to have an implementation faithful to the formal continuous semantics when the sampling interval's limit goes to zero. To use FRP in real-time applications, [48] presents Real-Time FRP (RT-FRP), a statically-typed language where the time and space cost of each execution step for a given program is statically bounded. As a followup to RT-FRP, [49] gives a formally verifiable compilation strategy from a new language based on RT-FRP into imperative code. Another language derived from RT-FRP is H-FRP which is more appropriate for hybrid reactive systems ([46]). H-FRP hides the details of sampling by providing a level of abstraction and also writing programs that diverge as the sampling rate increases becomes less likely.

1.2 Goals and Motivation

1.2.1 A DSL for Robot Programming

Our first main goal is to show how FRP can be employed to develop a domain specific language for robot programming and to demonstrate the capabilities of such a language. Before explaining our motivation behind this goal, let us look closer to the state of the art of robot programming.

The main issue of robot programming nowadays is producing appropriate software development environments and methodologies which will meet the demands of this domain. Hardware limitations or computing power are no longer constraints on developing robot control algorithms. Additionally the basic control algorithms for robots are well studied and there are libraries for such algorithms.

There are certain computational and domain characteristics of robot programming which will affect the design decisions of a robot control language.

Here are some of the specific features of robot programming resulting from being special real-time systems :

- Robots are supposed to *live* in the real world and the world is a chaotic and unpredictable environment where situation can change very rapidly. Therefore the robot should be reactive, dynamically configurable, responding quickly to cope with the world.
- Robots take their input from an analog environment so a robot programming language should have a way of denoting such continuous, synchronous data. The fact that data coming from the real world being noisy adds to the uncertainty and increases the emphasis on robust and self-sustaining programs.
- The final effect of robot computation will be on the hardware level. Robot languages should include methods to deal with these lower level computations in a way that these are hidden from the programmer.
- One common assumption in behavior-based robotics is the use of sensory-motor computation with multiple, tightly coupled loops. The robot language should have implicit or explicit methods to define such feedback loops.
- Robot computation is generally regarded as being composed of three layers. The low level is responsible for the feedback loops and deals with the hardware, ensuring the real-time reactivity of the robot. The top level is where planning is done to achieve a specific goal. The middle level connects these two by getting data from both of them and using this data to combine and sequence the tasks related to the goal. A robot language should address the needs of all these layers or alternatively, if the focus is on one of these layers, should have interfaces for the other layers.
- Robot programs are developed in an experimental way. They are continuously tested and improved. This calls for a language where rapid prototyping, software reusability and abstractions are crucial. Formal reasoning ability is also desired.

We believe that FRP meets all of these constraints. One can define complex behaviors dynamically evolving with time, and switch dynamically between different modes of computation to cope with the real world. FRP's

declarative style, strong type system, ability to develop modular, reusable, close-to-design, and high-level programs enables rapid prototyping. Abstractions can be easily developed depending on the task or system architecture chosen. There are already mechanisms and operators for intuitively defining feedback loops, hiding low level details and denoting continuous and discrete data. Domain-specific languages also have their own advantages. They have a localized semantics which means the host language's semantics is adapted to the domain instead of adapting the domain to the language's semantics. Software Engineering experience shows that it is easier to write, reason about, and maintain programs in a domain specific language.

A personal motivation was proving that functional languages are not only fun but one can develop serious real-world applications using them.

1.2.2 Hybrid Systems and FRP

The other main goal of this thesis is exploring the relationship between the formal specification of hybrid systems and FRP. We want to show how a formally specified hybrid system can be expressed in FRP and how FRP can be related to hybrid systems. For the latter, more specific questions are: given an FRP program can we find out a formal hybrid system specification associated with it, for what kind of FRP programs such a specification exists and can we find a mechanical way of getting this specification so that a transformer tool can be developed.

The importance of linking FRP to formal specification of hybrid systems and finding its weaknesses and strengths for expressing hybrid systems is twofold. First of all, if we know how to express a formally specified hybrid system in FRP then we can straightforwardly develop FRP programs for application domains which are hybrid systems and utilize the formal specifications already available for that domain. This will help designing and improving domain-specific languages with appropriate abstractions addressing the common notions of a particular domain.

Secondly, by getting a formal hybrid system specification of an FRP program, we can use the methods developed for analyzing hybrid systems for FRP programs. Algorithmic formal analysis of hybrid systems is a well studied area and by this transformation they will become available for FRP programs. It may be also possible to compare FRP and FRP-based domain specific languages to other frameworks and languages in terms of comparing their reductions to hybrid systems. Another outcome of relating code and specification in both directions is that by switching between these two and fine tuning them continuously, both of them can be improved to solve the problem in question. This is called round trip engineering and becoming more and more a desired feature of software engineering tools.

1.3 Approach

[38] and [39] describe Functional Robotics (Frob), a domain-specific language for robot programming developed by using FRP. Following the ideas in [38] and [39], we develop a dialect of Frob in the latest FRP implementation which solves some space leak problems occurring in the previous implementations. We explore the capabilities of Frob by comparing it to other recent robot control languages and developing a complex real-world application.

We also integrate abstractions for vision, specifically tracking, into Frob. We use the C++ library XVision2 [27] for this purpose. To find the right balance of glue and abstraction mechanisms for this integration becomes only possible after experimenting with the language. It is hard to know in advance importing what kind of objects and methods will be necessary for a problem domain. For instance, having a datatype Pixel in our language turned out to be useless since we do not need to develop vision algorithms in such a low level. On the contrary, we felt that it was best to import only the tracker components into Frob and develop the tracker abstractions in our framework to have more flexibility.

While such a fine tuning of our DSL goes on in the light of developing

the basic components and functionality for robot programming and vision tracking, simultaneously we investigated the most recent robot languages and architectures to see where Frob fits in the big picture. We will present a comparative review of these languages and Frob to show the pros and cons of using FRP for robot programming. This comparative review also helps improving Frob by adding abstractions to meet the requirements of the robot programming domain.

As a next step, a complex tracking-based robot navigation system is developed to check the practicality of Frob and to test it from the software engineering point of view. The problems encountered in this phase help to improve Frob more and to determine the right balance of integration.

To have a theoretical basis for FRP's usefulness we also investigate the relationship between FRP and hybrid systems. For this purpose we show that for a subset of FRP we can find a corresponding formal hybrid system specification which is faithful to the FRP program. We define a grammar for this subset and present a syntactic transformation from this grammar into formal hybrid system specifications. Based on this transformation procedure we develop a transformer where by employing the graph language dot [17] the final output is a graphical representation of the hybrid automata related to the input FRP program. For the reverse problem, we present methods and abstraction to mechanically define a FRP program corresponding to a hybrid system specification. We consider these as the first steps of creating analysis tools and a program development environment for complex software systems for robotics and other domains where FRP can be used.

Chapter 2

Functional Reactive Programming and Frob

In this chapter, we present *Functional Reactive Programming* and Frob, a domain specific language for robotics. There exist a number of concrete implementations of FRP. The latest full-scale implementation is called *Arrowized FRP* (AFRP) and this is the version that is used in our thesis work [33]. The following section gives an introduction to AFRP; the rest of the functions and abstractions are explained when they are used. We also define a layer of syntactic sugar that is employed in many of our examples. Since the current AFRP implementation is embedded in Haskell [41], its syntax is fundamentally that of Haskell. However, writing and reading AFRP programs is greatly facilitated by employing this layer of syntactic sugar.

2.1 AFRP

Two key notions 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 overall system input, or stimuli, and producing the overall system output, or response. The signal functions oper-

ate synchronously, in parallel, sensing a common *rate* of time flow. Another important aspect is that the structure of an FRP system can evolve over time. For example, new signal functions can be added or old ones deleted. Indeed, the entire network of signal functions could be replaced from time to time. 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 a *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*.

Any executable FRP implementation, like AFRP, can only approximate this conceptual signal model since continuous-time signals necessarily are evaluated for only a discrete set of sample points. The sample 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 preceding sampling point, or some interpolated value), but current FRP implementations are defined in such a way that this consideration is a non-issue.

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) [47]. 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 bandwidth deemed to be sufficient for the purpose at hand.

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
```

Thus, in AFRP, we can intuitively think of continuous-time signals as functions of type

$$\text{Time} \rightarrow A$$

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

$$\text{Time} \rightarrow \text{Event } A$$

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 `SF A B`. Intuitively, we have

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

If more than one input or output signal is needed, tuples are used for `A` and `B` since a signal of tuples is isomorphic¹ to a tuple of signals.

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

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 the result is a language which is much less susceptible to severe space and time leaks than some of the 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 [22]. A type constructor of arity two together with three operations, `arr`, `>>>`, and `first`, form an arrow provided certain algebraic laws hold. Ordinary functions are the canonical examples 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 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 point in time. `>>>` is a serial composition operator, similar to ordinary function composition. `first` is basically a “plumbing” combinator, making it possible to route signals in a network of signal functions.

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

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

This combinator composes two signal function 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. Supporting 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 [36] 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 ]
  pat1 <- sfxp1 -< exp1
  pat2 <- sfxp2 -< exp2
  ...
  patn <- sfxpn -< expn
  returnA -< exp
```

The keyword proc is analogous 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 *sfxp_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 *sfxp_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 (*sfxp_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 -< (c,d)

```

Here `sf1`, `sf2`, `sf3`, `sf4` are already defined signal functions. 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`. Figure 2.1 shows this signal network as a graphic.

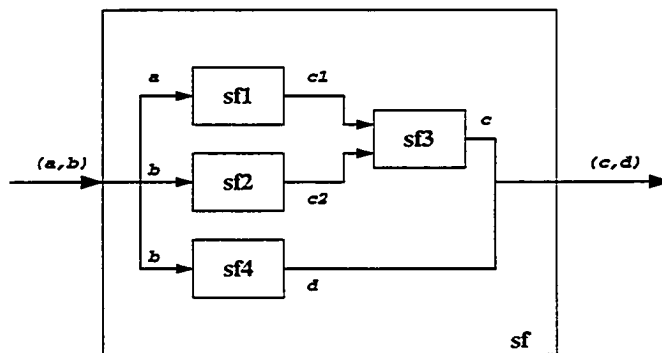


Figure 2.1: Visualizing the signal function syntactic sugar

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 boolean predicate supplied as the first argument.

2.2 Functional Robotics aka Frob

Functional Robotics (Frob), a domain-specific language for robot programming based on FRP, was first described in [38] and [39]. We develop a dialect of Frob in Arrowized FRP where the basic concepts are adapted to this new FRP implementation.

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 our presentation, we will target our Pioneer platform explicitly to keep things simple. The input and output types for the Pioneer robot are `PioneerInput` and `PioneerOutput`. Consequently, a top-level signal function for controlling the Pioneer has the type

```
SF PioneerInput PioneerOutput
```

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

```
reactivatePioneer ::
  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 use that 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.

2.2.1 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 [20]. They understood a task as a combination of a continuous behavior and a terminating event. They also noted that such tasks form a monad, allowing Haskell's monadic support 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 $\text{Task } A B C$, and we intuitively have

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

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 `taskexp1` and `taskexp2`. Switching from `taskexp1` to `taskexp2` 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 `taskexp3` is bound to the variable `x` and can thus be used in the expression `taskexp4`, 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.

Chapter 3

A Comparative Review of Robot Programming Languages

In this chapter, we make a comparative review of some recent robot languages and architectures. After giving short descriptions of the languages we look at the common features and discuss Frob in terms of these. For the languages which we can find example code, we show how to express them in Frob. We also show how some common notions of robot programming can be implemented in Frob.

3.1 Robot Programming

The appropriate languages, language structures, and software architectures for developing software for robot control 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) has emerged is not surprising, given the range of issues such a language would need to address.

One set of issues relates to the fact that an autonomous robot must inhabit an unstructured, dynamic, changing world, and therefore must be able to effectively react to its perception of the world based on only sensor information

and controlling only its 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 [7] or monitors [25] 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 [44] [43], where there is a logical separation between high-level planning, mid-level execution and monitoring of the steps of a task, 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 explicitly managing the flow of computation, while keeping code execution to the minimum necessary to perform the currently active computations.

3.2 A Review of Programming Languages

The term “robot programming language” encompasses a broad, disparate collection of work. At one end of the spectrum are languages¹ specifically designed for joint-level or Cartesian motion, e.g. RCCL [21] or Saphira [26]. 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 on the “intermediate-level” languages that have emerged in recent years, e.g. TDL [44] and Colbert [25]. 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 3.3 to demonstrate Frob flexibility.

3.2.1 Colbert

Colbert is called a sequencer language by its developers [25]. It is part of the Saphira architecture [26]. The Saphira architecture is an integrated

¹Here 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.

sensing and control system for robotics applications. Complex operations like visual tracking of humans, coordination of motor controls, and planning are integrated into 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 :

- Sequencing the basic actions that the robot will perform.
- 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 3.1 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 value 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 occurred.

```

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 3.1: Colbert Patrol and Approach Activities

Otherwise the loop terminates, suspends the patrol subactivity and the robot moves to the object.

In figure 3.2, 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`.

```
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 3.2: Colbert Patrol Activity with Interruption

Figure 3.3 shows how the patrol activity of Colbert in Figure 3.1 can be expressed in Frob. We first define a function to iterate a task for a given number of times (`iterateT`). By using this function we can repeat the patrolling task `n` times. A single patrol consists of moving ahead a given amount, turning back and moving in the reverse direction. The subtasks `turn` and `move` are used to define one tour of patrolling. For defining `turn`, first the heading angle is stored at the task initiation using `snapT`. If it is possible to send the robot a command for setting heading then it can be called with initial angle plus desired angle. Otherwise, the robot can start turning with a fixed velocity until the desired angle is met. Note that the type of `turn` enforces the input signal to have a field for odometry (because of `odometryHeading`) and the output

signal to have a field for setting translational and rotational speed of a differential drive robot (because of `ddSetVelTR`). `move` can be defined similarly. This time the position of the robot will be recorded when the task becomes active and the difference between the current position and the initial position will be checked to determine if the robot moved a given distance.

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

patrol1 = do move 1000 20
           turn 180
           move 1000 20

patrol n = iterateT n patrol1

turn :: (HasOdometry inp, HasDiffDrive out) => Double -> Task inp out ()
turn angle = do inp0 <- snapT
                mkTask (sf (odometryHeading inp0))
  where
    sf ang0 = proc inp do
      ang <- odometryHeading -< inp
      done <- edge -< (ang-ang0 >= angle)
      returnA -< (ddSetVelTR 0 5, done)

move :: (HasOdometry inp, HasDiffDrive out) =>
  Double -> Double -> Task inp out ()
move dist vel = do inp0 <- snapT
                  mkTask (sf (odometryPosition inp0))
  where
    sf pos0 = proc inp do
      pos <- odometryPosition -< inp
      done <- edge -< (pos-pos0 >= dist)
      returnA -< (ddSetVelTR vel 0, done)
```

Figure 3.3: Colbert Patrol Activity Defined in Frob

Expressing the approach activity in Frob demonstrates how existing tasks can be transformed to new ones for the purposes of a more complex task (Figure 3.4). The subtask `patrol1` in this example is first transformed to a

task which runs forever (i.e when it terminates it starts again). Then to this infinite task a terminating criteria is added by `objectFound` which is an event source occuring when an object is less than 2000 cm. This event's value is the distance of the object (the `tag` operator in the code is responsible for this). Note that `approach`'s type has the constraint `HasSonar i` because of the usage of `frontSonar` in `objectFound`. Other constraints are resulting from having `patrol1` as a subtask. Finally a timeout criteria is added, and depending on by which new criteria `patrol1` is terminated, the task returns either success after moving closer to the object or failure due to timeout.

```
data TaskResult = Success string | Failure string

approach :: (HasOdometry inp, HasSonar inp, HasDiffDrive out) =>
  Task inp out TaskResult
approach = res <- mkTask ((forEver patrol1) && objectFound 2000)
  'timeOut' 30
  case res of
    Left objDist -> do move (objDist-200) 20
                      return (Success "object approached")
    Right _       -> return (Failure "timeout")
where
  objectFound dist = proc inp do
    objDist <- frontSonar -< inp
    ev <- edge -< objDist<dist
    returnA -< ev 'tag' objDist
```

Figure 3.4: Colbert Approach Activity Defined in Frob

To implement the last Colbert patrol example involving interrupts in Frob is a bit more tricky since FRP does not have task signaling as a primitive and the number of patrols already finished is required to resume the task after an interruption. Firstly, we assume that `Task` monad is extended as a state monad to keep track of the number of patrols already finished as a task state. This information can be captured as part of the output signal of a patrolling subtask alternatively but using the monadic syntax will improve the readability, emphasis that this is a task state, and also extending any monad

as a state monad is straightforward. Secondly, since this example involves tasks signaling other ones via global channels for suspension or resume, we introduce such an abstraction for Frob such that our code will not be specific just to this example. We require that output signal has a field to record such requests and `reactimate` feeds these into the input signal to make every task aware of them at the next time step. Alternatively, without going deep into the `reactimate` level, a wrapper can be written which will extend a task's input and output signals with signaling commands by using `delaySF` for the same feedback effect.

In this Frob code (Figure 3.5), the classic state monad operations for initializing the state, reading the state, and modifying the state are denoted by the task operators `setState`, `getState`, and `modifyState` respectively. Note that, the type of the task has an extra argument for the state (`Integer` for this case). If no interruption occurs, by the help of these operators the sub-task `patrol1` will run for `n` times in the loop expression and the overall task will terminate successfully. If an interruption occurs, after the resume signal `patrol` will be called with a new argument which is the number of remaining patrols.

`checkSignalE` is used to listen to the global event signal for a specific task signal. Once the specific task signal is detected (i.e. another task sent this signal at the previous time step) an event is generated by `checkSignalE`. This scheme allows a task to respond to the signals that are issued for its name by simply capturing them as events. In fact, signals for any task can be detected if desired, by using that task's name in the `checkSignalE` argument.

3.2.2 Animate Agent Architecture

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 [15]. The basic datatype of the system are RAPs which are reactive plans defined in a Lisp-like syntax.

```

type TaskName    = String
data sendSignal   = Interrupt TaskName   | Resume TaskName
data receivedSignal = Interrupted TaskName | Resumed TaskName

checkSignalE :: HasInpSignaling inp =>
    receivedSignal -> SF inp (Event ())

patrol :: (HasOdometry inp, HasInpSignaling inp, HasDiffDrive out) =>
    TaskS inp out TaskResult Integer
patrol n = do setState 0
    res <- loop
    case res of
    Left _ -> res
    Right _ -> do stopT 'abortWhen' checkSignalE (Resumed "patrol")
        m <- getState
        patrol (n-m)
    where loop = (do modifyState (+1)
        patrol1
        m <- getState
        if m==n then return Success "done"
            else loop)
        'abortWhen' (checkSignalE (Interrupted "patrol") >>>
            arr ('tag' Failure "interrupted"))

```

Figure 3.5: Frob Patrol Code with Interruption

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 asynchronous 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 separately 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.
- 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.

```
(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)))
```

Figure 3.6: RAP task-net example

Figure 3.6 shows a task-net defined in RAP syntax. 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 [16] 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 asynchronous signals carried through global channels. `until-start` determines whether a task is started.

Firby gives the example in figure 3.7 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

```

(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 3.7: RAP Example

a condition to determine which methods will be activated.

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. Figure 3.8

shows how to implement this example in Frob.

```
approach = do camera ON
             res <- moveTo target |.| track target
             'abortWhen' (lost-target >>>
                           arr (tag LostTarget))
             'abortWhen' (camera-problem >>>
                           arr (tag Fail))
             'abortWhen' (stuck >>>
                           arr (tag Stuck))

             camera OFF
             case res of
               Left Success    -> ...
               Right LostTarget -> ...
               Right Stuck     -> ...
               Right Fail      -> ...
```

Figure 3.8: RAP Approach Example in Frob

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 the RAP system shows how to handle spawning tasks in subtasks. Unfortunately in the 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 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
```

3.2.3 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 [7] [8]. 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 than 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 AFSMs 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.

3.2.4 PRS

Procedural Reasoning System (PRS) is a general framework designed as a so-called situated reasoning system [24]. 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 [23] 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.

3.2.5 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 executable 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) [44] 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.

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 3.9 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. This task can be defined in Frob as in Figure 3.10.

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

```

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 3.9: A Task Tree for Mail Delivery

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.

3.2.6 SIGNAL

SIGNAL is a language designed for safe real-time system programming [19]. It is based on a semantics defined by mathematical modeling of multiple-

```

deliverMail room = do (x,y) <- getRoomCoordinates room
                      navigateToLocn (x,y)
                      centerOnDoor 'timeOut' 30
                      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 stopT

```

Figure 3.10: TDL Example in Frob

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 [13]. 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 preemptive 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

```

Figure 3.11: Signal Example for Mouse Clicks

The example in Figure 3.11 is taken from [19]. 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. Details of the notation may be found in the cited paper.

In Frob, by the help of the task monad and signal function syntactic sugar we can code this in a more intuitive way as in Figure 3.12.

`waitClick` waits for the first mouse click to switch to the task `afterDT` which will eventually terminate after specified time. The initial task result of `afterDT` is `Single` (set by `hold`) which changes to `Double` (again by `hold`) when a left mouse button click occurs.

3.2.7 Charon

Charon is a language for modular specification of interacting hybrid systems and can be also used for defining robot control strategies [3]. The building

```

datatype Result = Single | Double
mouse :: DTime -> Task GuiInput (Event Result) ()
mouse dt = do waitClick
              afterDT dt
              mouse dt

where
waitClick = mkTask (neverE &&& lbpE)
afterDT dt = let sf = proc inp do
              lbp <- lbpE           -< inp
              res <- hold Single -< lbp 'tag' Double
              t   <- localTime    -< inp
              ev  <- edge         -< t>=dt
              returnA -< (ev 'tag' res, ev)
            in mkTask sf

```

Figure 3.12: Signal Example in Frob

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.

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 constrained 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 clocks 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.

3.3 Frob Flexibility

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. This section illustrates this point by showing how Frob can be extended to cover some common robot programming paradigms. Like the examples in the previous section, it also shows the central FRP abstractions in use, and how higher-level abstractions can be derived in terms of these.

3.3.1 Finite State Automata

Languages like Colbert [25] and Charon [3] employ hybrid systems as their basic execution model. A hybrid system can be thought of as a collection of

continuous control modes and 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 [6, 25]. 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 3.13.

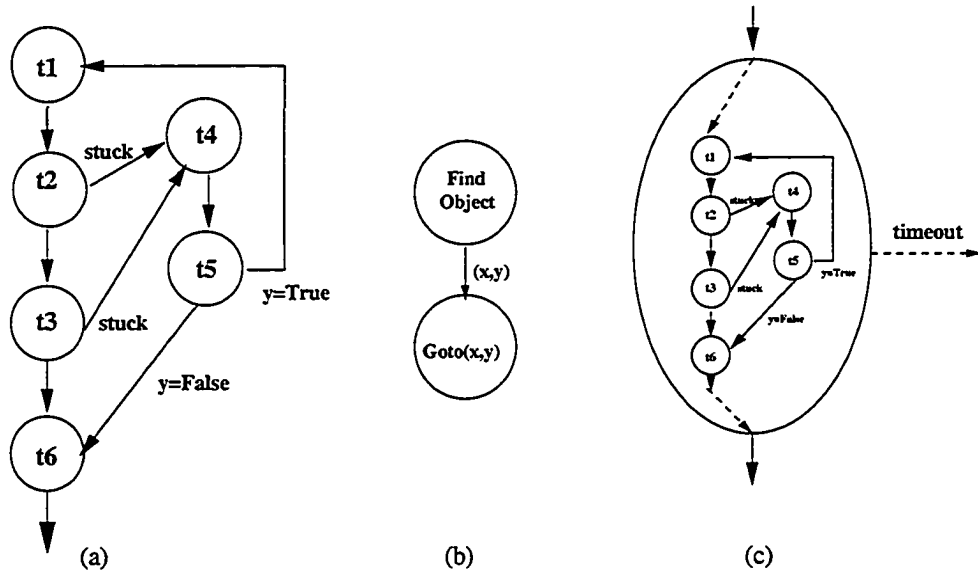


Figure 3.13: Expressing FSA's in Frob

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 3.13(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.

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

```

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

```

Figure 3.14: A task example

next mode; see Figure 3.13(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

```

Figure 3.13(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. 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 seamlessly with other task-based FSA encodings. This might seem as 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

```

3.3.2 Subsumption Architecture

The subsumption architecture is perhaps one of the most widely cited ideas for programming reactive robot systems [7, 8]. 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 convenience 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 it happen to be close, and travel to 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 to 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 have been

picked up. The pickup behavior's priority is 0 until the first pre-designated object is close. Then it raises to a sufficiently high priority, say 5, until the object has been picked, 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 successfully has been avoided, the priority of avoidance will drop and the other tasks regains 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. 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)
```

3.3.3 Assembling Behaviors

There are many other proposed behavior coordination mechanisms in robot programming [6]. They can be either cooperative or competitive. In competitive methods when two or more behaviors are active at the same time by using some kind of strategy one of them is selected as the current behavior. Cooperative methods alternatively uses more than one of active behavior's output to determine the current behavior. Here we give two examples to show how these different coordination mechanisms can be easily encoded in Frob.

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 = Double
datatype Plan = Avoid | Goto_Goal

choose :: [(Plan,Vote)] -> Behavior RControl
choose votes =
  let add = foldl 0 +
    sumV p = add (map (
  (p',v)->if p==p' then v
    else 0) votes)
    avoidV = sumV Avoid
    goalV  = sumV Goto_Goal
  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.

Here the highest priority is stayNear's. So if robot is away from the goal by more than maxGoal then it returns to the goal center immediately. If this predicate is not true then kickIt is checked as a candidate for being the active

```

type SC a = Behavior (a,Bool)

.+. :: SC a -> SC a -> SC a
sc1 .+. sc2 = (sc1 &&& sc2) >>> arr f
  where
    f ((a1, b1),(a2, b2)) = (if b1 then a1
      else a2, b1 || b2)

goalkeeper :: Behavior Point -> Behavior Point ->
Behavior RControl
goalkeeper pos ballPos =
  (stayNear .+. kickIt .+. blockIt) >>> arr fst
  where
    stayNear = travelTo goalCenter &&&
    (pos >>> arr f)
    where
      f p1 = distance p1 goalCenter > maxGoal
        kickIt = kick ballPos &&&
        ((pos &&& ballPos) >>> arr f)
        where
          f (p1,p2) = distance p1 p2 < maxK &&
            abs (vector2angle(p2.-.p1))<pi/2
            blockIt = (blocker pos ballPos) &&&
            (constant True)

```

Figure 3.15: Prioritized Behaviors for Behavior Merging

behavior. If the ball is closer to the robot than maxK and not at the corners of the playfield, then the robot goes to ball to kick it. If both of these constraints are false then the robot always tries to position itself between the ball and the goal since the constraint of blocker is always true. Here all of the functions travelTo, kick, and blocker are signal function with output signals of type RControl.

Most of the cooperative coordination methods based on behavioral fusion with vector summation. AFRP contains appropriate functions related to vectors and geometry. So these mathematical combination formulas can be effortlessly encoded in Frob.

3.4 Discussion

The table shows some of the different capabilities and design choices of the languages reviewed. 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 notion in these languages is the attention on providing a reasonable software development environment by enabling modularity and hierarchical decomposition. So we can say that the focus is shifting to the problems related to the software engineering dimension of robot programming. The papers on these languages generally stress and explain such capabilities of the language in question and consider this an important result.

	Anim.A.A.	Beh.L.	Charon	Colbert	PRS	Signal	TDL
1.	LISP	LISP	Special	C	LISP	Special	C++
2.	Task Nets	Augmented FSM	Hybrid Automata	FSM	Task Graphs	Data-flow Lang.	Task Trees
3.	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4.	T	B	B	T	T	B	T
5.	Ev.	Ev.	Ev.	Exc.	Ev.	Ev.	Exc.
6.	P	R,p	R,p	P	P	R,p	P

1:	Language Style	4:	(T)asks & (B)ehaviors
2:	Formalism Used	5:	Error Handling
3:	Concurrency	6:	(R)eactivity & (P)lanning

Figure 3.16: Language Comparison

Let's take a closer look at the features of these languages and make comparisons with Frob.

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. Robot programming requires two 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. The languages focused on specification of task scheduling mainly does not include modeling of continuous data (i.e. behaviors). Charon's behavior description mechanism is based on differential equations as being a hybrid system framework. Signal's is very close to Frob since FRP may be also seen as a synchronous data-flow language. Behavior Language's such mechanisms are based on the subsumption architecture. In these three languages there are ways to define tasks based on behaviors but none of them has a dedicated abstraction like Frob's.

Frob, like all of the languages, also focuses on modularity and building complex tasks and behaviors from simpler ones by its rich set of combinators and operators. Continuous data is represented by signals, an infinite stream of values of the same type. Asynchronous discrete events are represented by the Event type and proved to be very useful for modeling discrete mode changes of the system. As some of the examples demonstrate, task transformation operators and the Task monad enable to develop complex tasks for rapid prototyping. The languages in this review generally specialized in either specifying the behaviors or the tasks. Frob is the only language which is powerful at both of these two fronts by supporting modular specifications of both. As a result of this Frob programs can be detailed at both reactive and planning dimensions of robot programming but for the other languages either you have indirect support for one of them or no support at all.

Colbert, TDL, Animate Agents Architecture and PRS are mostly focused on planning and 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. The 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.

As required by the nature of robot programming, some form of concurrency is also available in all of the languages (i.e. multiple tasking in a single process or distributed processes). Frob's concurrency is based on merging signal functions –or tasks since the signal function part describes their behaviors– which modify the same output record type. This means when two different tasks are attempting to send commands to the same actuator, one of them will be ignored. The basic Frob design does not include global channels used for sharing data between tasks and/or making task interaction possible as in some of the reviewed languages. Nevertheless, additional abstractions can be developed to have the same convenience in Frob.

Error handling is another common feature of the languages. Exceptions and error related signals can change the control flow of the tasks. The event-driven approach where an error signal or a condition is checked as a task terminating criteria is more popular. As in RAP wait-for statement and Frob events the continuation may be also determined by the terminating event. 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 exception.

Formal verification of the robot programs are also desirable in the robot control domain. More formal languages like Charon, Signal, and Frob 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. Signal and

Charon have a special syntax appropriate for specifying signal networks and hybrid systems respectively. Other languages have also their own syntax for specification of a component and its relations to the other components (tasks or behaviors) but they have a very LISP-like flavor and may be considered domain specific languages with LISP as the host language.

The comparative review of these languages and Frob show that nothing fundamental is missing from Frob for being a robot programming language. This is good news, the better news is that Frob additionally have some key advantages. The FRP framework and the host language Haskell makes it very flexible in terms of choosing a certain robot architecture. The modularity and abstraction mechanisms provided by Frob enables rapid prototyping and experimenting with different strategies. As mentioned above, Frob also has the advantage of including both powerful behavior and task specification methods. A more detailed analysis of Frob is left to the final chapter which will make more sense after we present our work on developing a navigation system and looking into the hybrid systems relation.

Chapter 4

A Tracking-based Robot Navigation System

This chapter presents a tracking-based robot navigation system developed in Frob. To test Frob by developing a complex real world application is crucial because it is based on a functional language and the functional programming paradigm is conceived as not very suitable for practical purposes and may have serious performance problems. A navigation system based on visual input is a good choice as a testing ground since it reflects the problems and domain characteristics of robot tasks and robot programming. In this chapter, we also illustrate how notions from the application domains of robotics, vision, and user interfaces can be integrated within the FRP framework and discuss how FRP and Frob respond to the challenge of developing complex large-scale real world applications.

The developments in Computer Vision make it possible to use cameras as eyes of a robot and use vision mainly for sensing the environment. Algorithms for processing dynamic imagery is now at a point to be used in many application domains. There are libraries depending on efficient algorithms for basic problems. The focus is now on scaling these algorithms for complex problems, prototyping and evaluating experimental systems and effective integration of separate components. Therefore the FRP framework and declarative nature

of Haskell also becomes handy for vision applications.

For the visual component of our system, we employ XVision2 [27], a large library of C++ code for vision algorithms. XVision2 enables handling the low level, time consuming computation outside Haskell. XVision2 library provides generic interfaces to hardware components, like video sources and displays, a large set of image processing tools and, as an additional feature compared to similar libraries, it is more dedicated to vision tracking. Things in the real world may be in motion. Additionally, the static objects will appear to be in motion to a robot when it is moving. Hence visual tracking is an important part of using vision for sensing. A navigation system for robotics requires detecting the obstacles in the path, recognizing objects like walls and doors and track objects of interest for a particular task to be accomplished where navigation is part of. So we need a powerful visual tracking component for our navigation system.

It is not very clear what is the best level for integrating XVision2 to Frob. In particular, it is not required and desired to replicate all the class hierarchy of XVision2 in Frob. For instance, a Pixel object/type in Frob is not required since we will develop much higher level algorithms. On the contrary, we will definitely need to import tracker components. Images lie somewhere between these simple and much more complex objects. We will not implement image processing algorithms in Frob either but it turns out that we still may need to refer images (i.e. for initializing a tracker you need an initial image) so we have to include it in Frob. If we need two different image types for color and gray images in Frob is a more open question. These examples illustrate that we have to continuously fine grain the glue mechanisms and abstractions in Frob to find out the right level of vision integration for our purposes. It is also important not to be affected by XVision2's existing abstractions and try to develop our own which will take advantage of FRP and functional programming.

We took the approach of behavior-based robotics which is based on mapping stimulus to response for our navigation algorithms [6]. For our problem, using behavior-based robotics means that, rather than planning at a top level

the detailed steps of how to go from one point to another, the robot reactively determines its path depending on the current situation (i.e current sensor information) [28]. We assume that there is an imaginary force vector pushing the robot towards a certain direction and the robot uses this force vector to determine its current speed and direction. This force vector is computed dynamically taking into account where the robot is, what it sees and the task to be accomplished. For instance if the robot is trying to reach a certain point in space, the force vector will point from the robot's current position to the goal and its magnitude may reflect the distance. While navigating towards this goal, if the robot encounters obstacles in its path then the force vector should be modified by adding a component which pushes the robot against the obstacle(s) and dominates the goal vector when obstacles are too close. Similarly if there is a doorway ahead of the robot, and let's say the goal is behind and at the right of the door then the force vector should push the robot towards the door not to hit the right wall. After the door is cleared, the goal vector may become effective again and the robot will move to right.

Once such a force vector is computed, it is easy to calculate translational and rotational velocities to control a differential drive robot:

```
driveVector :: Vector -> (Velocity,Velocity)
driveVector (x,y) = (v,w)
  where
    a = radian x y
    v = limitA maxVelocity (k1 * x)
    w = limitA maxTurn (signum a * k2 * a * a)
    radian x y = f (if x == 0 && y == 0 then 0 else atan2 y x)
    f a = if a > pi/2 then a - pi else a
```

Here `limitA` is used to maintain the velocities below a threshold and `k1` and `k2` are constant gain factors. The only thing left is then to send appropriate driving commands to the robot with the velocities computed as above. In the next section, we show how vision related abstraction can be built in `Frob` and how visual information can be used to determine such a force vector.

4.1 Implementation

In this section we describe the basic functionality of our navigation system by walking through some code. In particular, we explain:

- our tracking abstractions and how to integrate them to the navigation system,
- how task abstraction can be used to specify complex navigation algorithms,
- and a simple yet useful user interface, designed within the same FRP framework.

We first turn our attention to visual tracking. Trackers usually 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. A predictor function may be used to modify the state before feeding it back to the source for better tracking.

```

type Src state = SF (ImageRGB, state) (ImageRGB, state)
type Stepper init state = init -> SF (ImageRGB, state) state
type Predictor state = SF state state
type Blob = (Int, Int, Int, Int)

```

In these type definitions `init` and `state` are polymorphic types which can be instantiated to types like `ColorSelector` or `Blob` respectively depending on the tracker they are components of.

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 ¹. 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` (figure 4.2 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 `delay` is used to close the loop. Figure 4.1 shows the interaction between various tracker components.

¹Native C++ methods can be called from inside Haskell by importing them by using the *greencard* Haskell foreign language interface.

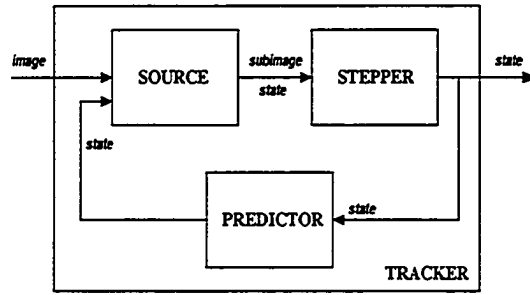


Figure 4.1: Tracker Components

```

type Tracker state = SF ImageRGB state

makeTracker :: state -> init -> Src state -> Stepper init state
              Predictor state -> Tracker state
makeTracker st0 init src step pred=
  proc img -> do
    (subImg, st') <- src      -< (img, st'')
    st          <- step init -< (subImg, st')
    st''        <- delay st0 -< st
    st'''       <- pred      -< st''
    returnA -< st

```

Figure 4.2: Tracker Abstraction in Frob as a Signal Function

The next step is to define a couple of tasks for interactive creation of trackers. Figure 4.3 shows how subtasks can be used to model the interaction with the user for the creation of a tracker.

We give these definitions in detail to show how task abstraction can be also useful for the vision domain. `getClick` is a very simple task which outputs no graphics and terminates with a left mouse button click with the value of the mouse position. 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 changing 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

```

getClick :: RobotTask (Int, Int)
getClick = mkTask (st &&& fvgiLBP)
  where st = constST (fvgoOverlaySimpleGraphic SGEEmpty)

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 (Tracker Blob)
colorTracker = do
  blob0 <- getBlob
  inp    <- snapT
  let c = getColorSelector (subImage ((fviImage inp),blob0))
      track = makeTracker blob0 c source1 colorStepper (arr id)
  return track

```

Figure 4.3: Creating Trackers with Tasks

position as the second corner of the blob ($c2$). `getBlob`'s task result is the blob which can be computed from $c1$ and $c2$. In `colorTracker`, a snapshot is taken of the 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 ability 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
```

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 by some upper values to prevent the robot from moving too fast.

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 4.4 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`. Figure 4.5 shows the different modes of the system based on these events.

The `XVision2` function `obsLines` returns a list of lines which are projections of obstacles ahead. Together with the position of the tracked object, these lines are used by `toVector2` 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

```

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 -< lMergeEvents e1 (lMergeEvents e2 e3)
butSG = drawBlob but1 'mrMerge' drawBlob but2 'mrMerge' drawBlob but3
  'mrMerge' fvgiOverlayText (20, 25) "COLOR TRACKER" Red
  'mrMerge' fvgiOverlayText (15, 55) "STEREO TRACKER" Red
  'mrMerge' fvgiOverlayText (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 fvgiOverlayText (160, 20) str Red

```

Figure 4.4: Main task for Tracking-based Navigation System

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

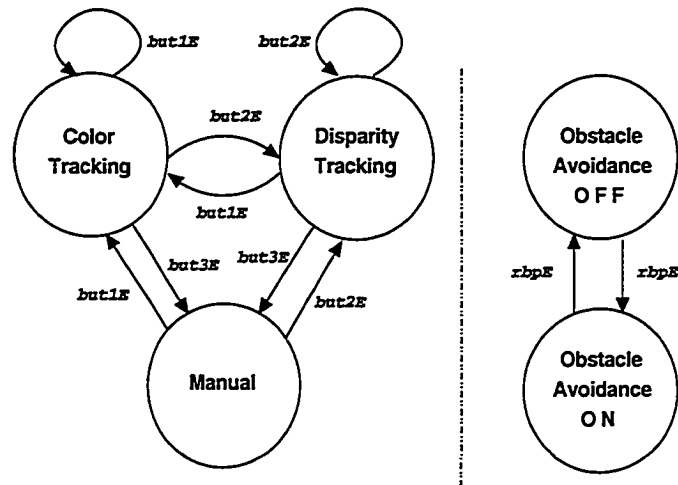


Figure 4.5: Mode Switchings of the Navigation System based on Events

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 4.6 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`.



Figure 4.6: User-interface while robot is navigating towards the tracked object

This example shows how one can define modular, hierarchically decomposed thus appropriate for rapid prototyping and experimenting with robot control software in Frob.

For instance, consider these modifications and additions to the navigation system we present:

- Adding an estimation algorithm to the source of the tracker to determine more correctly the location of moving objects in the next image.
- Changing how the user specifies a blob. For instance, instead of choosing a rectangle, a mouse click will be enough and the system will find a reasonable blob containing the mouse position.
- Adding more trackers with different steppers.
- Defining a more general tracker which will switch between different trackers based on their performance. The rest of the system will not be aware of which specific tracker is in use currently. The only criteria is that these trackers' state have the same type.
- Improving the navigation algorithm by adding more cases to the system.
- Modifying the user interface. This can include changing how it looks or how it is used independently following the Modal-View-Controller GUI design pattern.
- Additions to user interface reflecting the added functionality to the navigation algorithm.
- Displaying some data (i.e. distance to the tracked object) related to the task.
- Changing how force vector is computed to improve the quality of the navigation.

These additions and modifications span from little details to more top level specifications and still it is very clear which part of the code will be worked on. It is also ensured that plugging the new component will not affect the rest of the system as long as it type checks correctly and its implementation is faithful

to its role in the system. When something goes wrong, strong type checking, declarative style of the programs and clear distinctions between the system components helps the programmer to find if the problem occurs because of the design or the code and also the part of the code which is the problem source.

Chapter 5

Hybrid systems and FRP

In this chapter, we investigate the relationship between FRP and hybrid systems. First we describe how a formally defined hybrid system can be expressed in FRP. We give some abstractions to show that this can be done mechanically. Then we look at the reverse problem. Given an FRP program how we can determine a corresponding hybrid system. We give a translation procedure showing that, for a reasonable subset of FRP, we can find such a specification. This proof also makes it clear what kind of FRP programs are not directly expressible as a hybrid system specification. We give a grammar to distinguish such FRP programs and also to present the translation without mentioning the technical details of using FRP in a host language. This way we can concentrate on the FRP constructs independent of the host language it is implemented. A syntactic translator based on this procedure and written in Haskell is given at Appendix A. The translator can be also used to display FRP programs as a graph (by the use of dot graph language) where the graph represents the formal hybrid system specification.

The difficulties of translation of a formal hybrid system into FRP code and finding a formal specification for any given FRP code are different. Although in theory both are Turing equivalent, the former is more straightforward than the latter. There is not a consensus on the formalism that suits best the hybrid systems. The most common one, which we also employ here, is based

on finite state automata. It is suitable for formal analysis but can not capture all the possible hybrid systems just like FSA's can not capture all kinds of computations. Since FRP has all the expressive power of a functional language the situation for the two kinds of translation is not symmetrical. We can give an corresponding FRP program for any hybrid system defined in the formalism we use but the reverse is not true. We can even have an abstraction in Haskell resembling the formal specification and produce FRP code that will implement the hybrid system automatically. On the other hand, we are forced to define a proper subset of all possible FRP programs in order to give a translation procedure that will give us the formal specification in detail.

Every FRP program defines a hybrid system in the general sense but the expressive power of FRP allows to define hybrid systems which can not be captured as a formal specification. The higher-order programs that can be defined by the rich set of FRP combinators and operators can not be directly transformed into a static formalism. Higher orderness can result in dynamic systems in many different levels (i.e. for Frob, both dynamic behaviors and tasks). In theory, it is possible to eliminate the higher orderness and dynamic components of a FRP program. There are many well studied ways to do this in the functional language literature. The equational reasoning of the functional programming paradigm makes automated systems achievable that will syntactically transform programs for a desired end result. The problem with this approach is that the structure of the original program gets lost during the transformation. In some way, this is similar to converting a NFA to a DFA where semantics is preserved but the structure is obscured. We want to apply formal analysis techniques to the FRP programs by the help of such a transformation but we also want to use the formal specification of the hybrid system for software engineering purposes. So we choose finding out a transformation procedure for a subset of FRP programs where the resultant hybrid system will faithfully represent the structure of the FRP program. This will also help pointing out the shortcomings of the current hybrid system formalisms and may lead to defining more expressive ones in sync with the latest hybrid

system specifying tools and environments.

5.1 A formalism for hybrid systems

We first present a formal definition of a hybrid system adapted from [31]. A hybrid system is a structure $\langle V, X, \Rightarrow, \varphi \rangle$ where:

- $V = \{v_1, \dots, v_m\}$ is a finite set of vertices (or locations);
- $X = \{x_1, \dots, x_n\}$ is a tuple of state variables, ranging over some domain D_X ;
- $\Rightarrow$ is a subset of $V \times B(X) \times F(X) \times V$, denoting edges, where:
 - $B(X)$ is the set of predicates on X , that is, if b in $B(X)$ then b is a mapping from D_X to $\{true, false\}$
 - $F(X)$ is the set of internal mappings of D_X , that is, if f in $F(X)$ and X_0 in D_X then $f(X_0)$ in D_X .
- $\varphi : V \rightarrow [(D_X \times R^+) \rightarrow D_X]$ is the evolution function.

In this extended finite automata specification a set of state variables (X) are used to trace the state of the system. The evolution functions (φ) on these state variables describe how the system evolves. Different behaviors of the system are modeled by having different locations or vertices (V). Note that an evolution function of a state variable both depends on a location and a positive real value which is time. We will write $\varphi_{v_i}(x_i)(t)$ denoting the evolution function value of the state variable x_i at location v_i and time t . The phase transitions of the system from a behavior to another behavior are modeled by edges between the vertices. These edges ($\Rightarrow$) are defined by predicates on state variables ($B(X)$) where the system changes behavior when the predicate becomes true. Moreover edges may include value assignments ($F(X)$) for some state variables initializing their values for the new location.

We can say that hybrid systems are composed of different modes of computations. While the system is in one mode the computation has a continuous characteristics. The locations and the evolution functions of the formalism captures this property of the hybrid systems. The state variables evolve by the same set of rules while a mode/location is active. Then there may be discrete changes in the system's behavior where the mode of computation changes. The phase transitions of the formal specification defines such discrete changes. At a certain point more than one of the transition predicates may be true which means hybrid systems can be non-deterministic. The information flow between two different modes of computations are modeled by the value assignments $F(X)$.

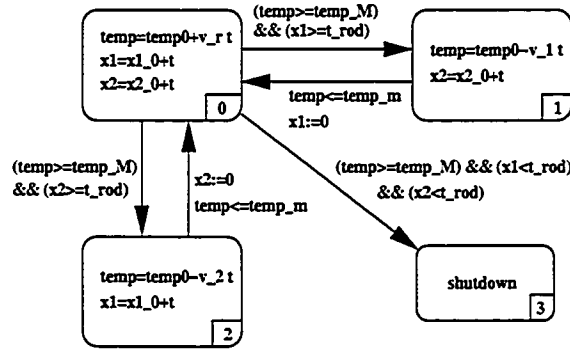


Figure 5.1: Hybrid system for the temperature control system

Consider the example in Figure 5.1, taken from [31]. The temperature control system in this example controls the coolant temperature in a reactor tank by moving two independent control rods. The system maintains the coolant temperature between temp_m and temp_M . If the the temperature reaches temp_M then one of the rods is used to refrigerate the tank. However, a rod cannot be used until t_{rod} units of time passes after its last use. If neither rod is available when temperature reaches temp_M a complete shutdown is performed.

As the figure shows, we use three state variables to model this control system. temp denotes the temparature, $x1$ and $x2$ are counters to keep track

of how much time passed since the last usage of two rods. Although the temperature is an outside factor we have to model the transformation of its value through time in our specification to enable formal analysis like if the shutdown state is reachable for certain state constants. The more faithful our modeling of the temperature, the more dependable will be the results of the formal analysis.

In the initial location 0, the evolution functions tell us that the temperature increases with a rate of v_r and both rod counters are incremented. The locations 1 and 2 correspond to the system behavior where rod 1 or 2 is in action respectively. The temperature decreases by a rate of v_1 or v_2 depending on the rod used and the counter of the unused rod still increases. The counter of the used rod is ignored since its value will be initialized to 0 when the temperature drops to $temp_m$ and a phase transition will put the system in location 0 again.

The formal specification of this hybrid system will be:

- $V = \{0, 1, 2, 3\}$
- $X = \{temp, x1, x2\}$
- $\Rightarrow = \{$
 - $(0, (temp \geq temp_M) \ \&\& \ (x1 \geq t_rod), \ \{\}, \ 1),$
 - $(0, (temp \geq temp_M) \ \&\& \ (x2 \geq t_rod), \ \{\}, \ 2),$
 - $(0, (temp \geq temp_M) \ \&\& \ (x1 < t_rod) \ \&\& \ (x2 < t_rod), \ \{\}, \ 3),$
 - $(1, (temp \leq temp_m), \ \{x1 := 0\}, \ 0),$
 - $(2, (temp \leq temp_m), \ \{x2 := 0\}, \ 0) \ \}$
- $\varphi_0(temp)(t) = temp0 + v_r * t$
- $\varphi_1(temp)(t) = temp0 - v_1 * t$
- $\varphi_2(temp)(t) = temp0 - v_2 * t$
- $\varphi_0(x1)(t) = x1_0 + t$
- $\varphi_2(x1)(t) = x1_0 + t$
- $\varphi_0(x2)(t) = x2_0 + t$
- $\varphi_1(x2)(t) = x2_0 + t$

where t denotes time and $temp_m, temp_M, v_r, v_1, v_2, t_rod$ are system constants.

5.2 Expressing Hybrid Systems in FRP

In this section, we describe given a hybrid system specification, how an FRP program can be developed which will implement the behavior of the system. We present a general scheme which is independent of the details of the specified hybrid system. An abstraction can be used to define hybrid systems as the extended finite automata described at the previous section and turn this definition into an FRP task. We will use the temperature control system as an example to explain the general strategy.

We first discuss how to define a signal function corresponding to a single location of the system. This signal function emits a pair of signals: one consisting of the values of the state variables, and second one is an event source defined by a combination of outgoing edge predicates. The value of the event carries the current values of the state variables when a transition is enabled. So these values can be used for the initialization of the next location. Note that time can be accessed by using the primitive signal function `localTime` on the input signal. Putting these together, here is the FRP code for location 0 of our example:

```
loc0 temp_0 x1_0 x2_0 =
  proc inp -> do
    t <- localTime -< inp
    let temp = temp_0 + v_r * t
        x1   = x1_0 + t
        x2   = x2_0 + t
        stateVars = (temp, x1, x2)
    tr1 <- edge -< temp >= temp_M && x1>=t_rod
    tr2 <- edge -< temp >= temp_M && x2>=t_rod
    tr3 <- edge -< temp >= temp_M && x1<t_rod && x2<t_rod
    let e = mergeE [tr1 'tag' "loc1", tr2 'tag' "loc2",
                  tr3 'tag' "loc3"]
        pred = e 'tagE' (\loc -> (loc, stateVars))
    returnA -< (stateVars, pred)
```

Here `temp`, `x1` and `x2` are the state variables. `tr1`, `tr2` and `tr3` denote the outgoing edges of the location. `edge` is used to convert the boolean signal (here

the edge predicate) to an event source where the event occurs each time the boolean signal changes from False to True. Each event corresponding to an edge is tagged by a string that denotes the next location, and these events are combined with `mergeE` to form the overall predicate for the location. Note that the final output event value `pred` includes both the next location information and the values of the state variables (added by `tagE`) when the event occurs.

After defining a signal function for each location like this, we can turn them into FRP Tasks in order to define the data flow between the vertices. The initialization of the next active location can be also handled while defining these relations since the result of a task is determined by the terminating event and the events are tagged for next location and the final values of the state variables:

```
t0 temp_0 x1_0 x2_0= do
  (next,(temp_0,x1_0,x2_0)) <- mkTask (loc0 temp_0 x1_0 x2_0)
  case next of
    "loc1" -> t1 temp_0 x1_0 x2_0
    "loc2" -> t2 temp_0 x1_0 x2_0
    "loc3" -> t3 temp_0 x1_0 x2_0
```

The triple `(temp_0,x1_0,x2_0)` is the final value of the state variables at termination and is used for initializing the next location. The event value is used to determine the next location by the case expression. `t1`, `t2` and `t3` denote the tasks corresponding to the other locations of the temperature hybrid system; these are defined in a manner analogous to `t0`.

We can develop an abstraction for representing hybrid systems based on this general strategy. Figure 5.2 shows this abstraction and as seen it is very similar to the formal definition of a hybrid system. A `HybridSystem` consists of a list of triples where each triple defines a location of the hybrid system. The type `st` is a tuple used for denoting the state variables. So if the hybrid system has variables x and y then the polymorphic type `st` will be instantiated as the pair (x,y) . The first argument `Location` is a string which denotes the name of the location. The second argument is a function returning the signal function which defines the evolution functions over the variables in that

```

type Location          = String
type Transition st     = (Location, st -> Bool, st -> st)
type HybridSystem inp st = [(Location, st -> SF inp st,
                             [Transition st])]

runHS :: Location -> HybridSystem inp st -> st ->
      Task inp st (Location, st)
runHS loc0 hs init0 = do let t = lookup loc0 hs
                        (loc, init) <- t init0
                        runHS loc hs init

where
lookup loc [] = error "State not found"
lookup loc ((loc1, st1, trs):rest) =
  if loc==loc1 then
    \init ->
      let f (loc, pred, init) = proc state -> do
                                e <- edge -< pred state
                                returnA -< e 'tag' (loc, init state)
          tr = mergeEs (map f trs)
          t = proc inp -> do
              state <- st1 init-< inp
              e <- tr -< state
              returnA -< (state,e)
      in
        mkTask t
  else lookup loc rest

```

Figure 5.2: An Abstraction for Hybrid Systems

location. The argument of this function, `st`, is used for the initial values of the variables. Then the last value of the triple is a list of outgoing transitions. Each `Transition` is also a triple defining a subsequent location, the predicate function over state variables corresponding to the edge transition conditions, and another function over variables to convert state values into values that will be used to initialize the next location. If the final values are used unaltered, then this function is simply the identity function.

After defining the hybrid system in this way, `runHS` can be used to convert the system into an FRP Task. `runHS` takes a initial location, the hybrid system

```

temp_control =
let (v_r, v_1, v_2) = (0.5, 0.2, 0.3)
    (temp_M, temp_m, t_rod, temp0) = (50, 30, 100, 35)
in
  [ ("S0", \ (temp_0,x1_0,x2_0) -> proc inp -> do
      t <- localTime -< inp
      let temp = temp_0 + v_r * t
      x1  = x1_0 + t
      x2  = x2_0 + t
      returnA -< (temp, x1, x2)
    , [ ("S1", \ (temp,x1,x2) -> (temp>=temp_M && x1>=t_rod), id),
        ("S2", \ (temp,x1,x2) -> (temp>=temp_M && x2>=t_rod), id),
        ("Shutdown", \ (temp,x1,x2) -> (temp>=temp_M && x1<t_rod
            && x2<t_rod), id)
      ]
    ),
    ("S1", \ (temp_0,x1_0,x2_0) -> proc inp -> do
      t <- localTime -< inp
      let temp = temp_0 - v_1 * t
      x1 = undefined
      x2 = x2_0 + t
      returnA -< (temp, x1, x2)
    , [ ("S0", \ (temp,x1,x2) -> temp<=temp_m,
        \ (temp_F, x1_F, x2_F) -> (temp_F, 0, x2_F))
      ]
    ),
    ("S2", \ (temp_0,x1_0,x2_0) -> proc inp -> do
      t <- localTime -< inp
      let temp = temp_0 - v_2 * t
      x1 = x1_0 + t
      x2 = undefined
      returnA -< (temp, x1, x2)
    , [ ("S0", \ (temp,x1,x2) -> temp<=temp_m,
        \ (temp_F, x1_F, x2_F) -> (temp_F, x1_F, 0))
      ]
    ),
    ("Shutdown", undefined, undefined)
  ]

```

Figure 5.3: The Temperature Control Example

and the initial values of the variables as its arguments. The current location is converted into a task which will terminate either one of the edge predicates

becomes true and the location activity is defined by the signal function of the task. Since this terminating event also returns the next location and the initial values, when a transition occurs a task can be generated on the fly corresponding the next location and the overall task switches to this task. The local function lookup takes a location (which is initially the argument `loc0`) and the hybrid system and generates the task corresponding to that location. Note that the tasks created by `runHS` can also be converted back into a signal function and merged with other hybrid systems or likewise used as a super node.

Figure 5.3 shows the complete temperature control example defined by using the FRP abstraction. It is worth noting that in practice some syntactic sugar may be defined to make the creation of this form more elegant. For instance, with adding a declaration of the state variables at the top, most of the redundancy will go away. The strong type system of Haskell prevents us to do so elegently without a syntactic sugar.

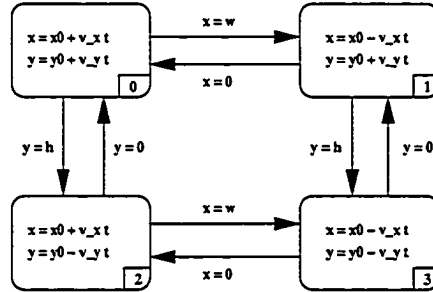


Figure 5.4: Hybrid system for the movement of a billiard ball

Since this is a general strategy, the code we give for the temperature control system is not the best way to describe it in FRP. In a functional language there are much better ways to encode models like this but in this case we want our abstraction to be similar to the formalism we employed so one can directly encode a formal specification and then automatically get the corresponding hybrid system.

Consider another example from [31] given in Figure 5.4. This hybrid system

describes the position of a billiard ball through its state variables x and y . Assuming the bottom left corner of the billiard table is at position $(0,0)$ and the upper right corner is at (w,h) , we can define predicates corresponding to ball's hitting an edge of the table. When such a predicate becomes true, the ball's vertical or horizontal speed changes sign and the system moves into a new state reflecting this change.

```
ball x0 y0 v_x v_y w h =
  proc inp -> do
    x <- integral -< xDir * v_x
    y <- integral -< yDir * v_y
    hitX <- edge -< x<=0 || x>=w
    hitY <- edge -< y<=0 || y>=h
    xDir <- stepAccum 1 -< hitX 'tag' (\x -> -x)
    yDir <- stepAccum 1 -< hitY 'tag' (\x -> -x)
    returnA -< (x0+x, y0+y)
```

Figure 5.5: Billiard Ball Hybrid System in FRP

If we use our general strategy to define this hybrid system in FRP depending on this specification, we will end up with 4 signal functions and 4 tasks but since the evolution functions are similar in each location we can capture the behavior of the system in FRP much more elegantly and intuitively (Figure 5.2).

This example illustrates how the declarative power of FRP can be useful to describe hybrid systems but the advantages of FRP for this purpose is not limited to its functional style. For instance, in our navigation code we are using a tracker task to employ a tracker for a specific object to compute the path of the robot. This is an example of hierarchical modeling and reusability which is hard to capture in this formal specification of hybrid systems. The high level programming capabilities of FRP can result in programs which are intuitive and elegant and yet outside the realm of a simple formal specification. In the next section, when we define what kind of FRP programs can be mechanically converted into hybrid system specifications, this issue will become more clear.

5.3 FRP programs as a Hybrid System

What is more interesting than showing how hybrid systems can be represented in FRP is to determine the hybrid systems structures of FRP programs. To what extent can a hybrid system be developed which is equivalent to an FRP program? What constructs of FRP make this transformation difficult?

Since there can be more than one hybrid system specification where the system they define behaves equivalently, first of all we have to answer the question that in what detail we want our hybrid system specification to be. For instance, the evolution functions can be defined as the composition of partial functions. Thus, any FRP program can be defined as a single state hybrid system where, with a sufficiently complex evolution function, the output is computed from the input in each time step. The mode switchings in the FRP program will be hidden in the hybrid system specification with this kind of representation. They might be deduced by inspecting the evolution functions but by no means we can say this is a faithful representation of the FRP program as a hybrid system. So it makes much more sense to have a state transition in the specification for each switching term in the FRP program. A similar problem is about the decision of which set of state variables will be included in the specification. We can define a single variable just representing the output of a signal function but again we will not make explicit how this value is computed through a signal function network. The specification will be more faithful if the set of state variables and corresponding evolution functions reflect the programmer's labeling of intermediate signals and choice of signal function components in a complex signal function network.

We first give a grammar to distinguish the subset of FRP programs that can be faithfully expressed as a hybrid system (Figure 5.6). This grammar decouples the FRP constructs from the host language, making it easy to concentrate on the reactive properties of an FRP program. We will call the language corresponding to this subset of FRP as HS-FRP (HS for Hybrid Systems or Hybrid Specification).

```

sf ::= fst sf | arr ext_fn | seq sf sf | delay ext_val | const ext_val |
      sf switchev cont_sf | edge | localTime | integral |
      let-sig lab = sf in sf | lab |
      sf-net (patr, [let], [sig-map], exp)
sig-map ::= (patr, sf, exp)
let ::= let patr = exp
patr ::= var | (patr, patr)
exp ::= var | (exp, exp) | ext_fn exp | ext_val | ev
ev ::= var | mergeE ev ev | tagE ev exp
cont_sf ::= sf | lambda.patr sf | lambda.patr if ext_pred sf sf

```

Figure 5.6: Grammar for HS-FRP

An FRP program is mainly about specifying an ultimate signal function which will describe the reactive behavior of the system. Therefore our grammar focuses on how signal functions can be defined. We include in our grammar *sf-net*, corresponding to the syntactic sugar definition of a signal network for a more clear presentation of our proof. In a *sf-net* definition, *patr* denotes the input signal, *exp* describes the output signal and intermediate signal mappings are captured by the *sig-map* terms. As in the original syntactic sugar, there may be *let* expressions to eliminate redundancy and to improve readability. Here is a simple signal function example and how it can be defined as a *sf-net* in our grammar:

```

sf-net ((a,b), [let pi = 3.1415], [ (c, arr f, a), (d, delay 0, 2 * pi * b) ], c+d) ≡

sf = proc (a,b) -> do
  let pi = 3.1415
  c <- arr f -< a
  d <- delay 0 -< 2*pi*b
  returnA -< c+d

```

Note that any signal function *sf* not defined by *sf-net* can be expressed as (*inp*, [], [(*out*, *sf*, *inp*)], *out*) where *inp* and *out* are dummy variables.

First important observation on this grammar should be the usage of *ext_fn* and *ext_val* which sets the reactivity of an FRP program in the context of a host language. *ext_val* denotes values coming from the host language (i.e 0 in the

delay expression of our example is a value exported from the host language). In fact, this can be any base language expression that is not including a HS-FRP variable (labelings for signals). *ext_fn* denotes external function calls to the host language¹. These external computations help to create new signals whose values at each time instant depend on the already defined signals pointwise. The new signal may be an intermediate signal as a result of an *arr* term (i.e. c in the example), the final output signal of a signal network (i.e. $c+d$ in the example), or an input signal in the signal network (i.e. $2*pi*b$ in the example).

We are interested in the *syntactic* translation of HS-FRP programs into a hybrid system specification so for the scope of this thesis we do not address the *semantic* properties of this bridge to the host language (i.e if these external function calls terminate or time/space complexity of the computation). Nevertheless, when these specifications will be used for formal analysis, there should be some semantic constraints (modulo the host language) on these external calls for the soundness of the formal analysis. These constraints will depend on what kind of formal analysis is desired.

Since our goal is syntactic translation, we omit a type system and operational semantics for our grammar. RT-FRP's operational semantics [48] can be adopted to our grammar. This is trivial in the sense that two grammars *almost* define the same set of programs (we will look at their relation more closely in the next section). It is also possible to define a type system for HS-FRP based on the Haskell types of signal functions and the usual type judgments. The only thing crucial and specific to the type system of our grammar will be related to the *tagE* terms which modifies an event's value. We require that events carry only values of the base language (i.e. integer, real, boolean). The reason behind this constraint is related to the the main role of the events in FRP: defining the mode switching points and parameterizing the continuation in the system. Recall from the previous section how we utilize events for defining FRP programs corresponding to a formally defined hybrid system.

¹Tuples can be used for curried functions with more than one argument

We modify the events' value both for determining the next location of the system and also to carry the state variable information to the new location. We want to limit this expressive power with our typing rule since otherwise, if we allow higher order event values that can be used to instantiate the new active location, it becomes very hard to give a realistic definition of the new location.

To give an example, consider the following signal functions that emit an event signal:

```
cnt = edge >>> arr (\e -> tag e (\x->x+1)) >>> stepAccum 0
sf  = edge >>> arr (\e -> tag e (\f->f.f)) >>> stepAccum f
```

Here `stepAccum` is a step accumulator which outputs its argument until the event in its input signal occurs. This event's value should be a function that is applied to the current output to get the new output signal value, which remains constant until the event occurs again. Then same procedure is applied to determine the new output value. So `cnt` is a counter which starts from 0 and increments every time the event defined by `edge` occurs since that event's value is modified to be the increment by one function before feeding it into `stepAccum`.

Second example is quite similar to the counter example but this time the initial output value is a function f and this output function is doubled every time the event defined by `edge` occurs (". " is the function composition operator in Haskell). If we allow events to carry function values, then writing signal functions like sf will make it possible to have function variables whose value change every time an event occurs. For the generally accepted formal hybrid system definitions this means evolution functions in a location can change dynamically and it is contradictory to the notion of evolution functions. One can argue that this is in fact a discrete change in the system triggered by an event and we can clone locations with evolution functions containing functions f^n with different n values but since there is no upper bound for occurrences of an event to do this we will need an infinite number of locations. Such a hybrid

system definition will be also counterintuitive. Events returning signals or other events will result in more controversial hybrid systems even if somehow they can be defined.

In fact, as in RT-FRP, the basic limitation of this grammar is not explicit from the rules. The fact that the reactive part is decoupled from the base language is responsible for most of the restrictions. Such a subset of FRP allows only composing signal functions through the reactive components and reduces the role of the base language mainly to manipulate the mappings between input and output signals. An external call to the base language can not return a signal function.

If we can use the full power of the base language, we can express signal function compositions like:

```
sf_seq n sf = if n=1 then sf
              else sf >>> (sf_seq (n-1) sf)

mode-seq n sf = if n=1 then sf
                else sf 'kSwitch' (\(m,sf') -> if even m
                                                then mode-seq (n-1) sf
                                                else mode-seq m sf')
```

In the first example, we can define a sequence of a signal functions where the length of the sequence depends on the number n . Suppose that the signal function to be sequenced is simply $\text{arr } f$, then this sequence will define the signal function $\text{arr } (f^n)$ which as discussed above may be problematic. When the sequenced signal function is more complex what the resultant sequence can mean as a hybrid system becomes even less clear.

The second example is more complex and controversial. In this one, a signal function is sequenced by switching. Depending on the event's value that is causing the switching, either a switch occurs within the sequence (if m is an even number) or a whole new sequence is created on-the-fly depending on a new signal function and with a dynamically determined length where

both comes from the event value. This sequence may give way to another sequence when the new signal function's terminating event returns an odd value. It is very hard, if not impossible, to capture the behavior of recursive signal functions like these in the context of a hybrid system specification. As mentioned in the introduction the higher orderness of FRP results in such dynamic definitions. If we have static forms of similar signal functions (i.e. constant n in `sf_seq`) then we can give a corresponding formal definition.

HS-FRP grammar also disallows signals with signal values since a signal's value can be manipulated by external base language calls only. This again prevents to express signal functions which will be hard to specify as a hybrid system but also rules out some higher level signal functions which can be quite useful. For instance, suppose that in our navigation system we want to track more than one object. We want to add a tracker automatically for each moving object that enters into the sight and delete the corresponding tracker when an object becomes out of the sight. The best way to express the set of active trackers in the system will be maintaining a signal of a list of trackers. At each time step, the value of this signal will enable to access the active trackers. Since trackers are also signal functions, this nice modeling will be illegal in HS-FRP.

These limitations of HS-FRP suggest that we need a better way of formally specifying hybrid systems. Such a specification should both allow formal analysis and also capture the power of frameworks developed for specifying hybrid systems like FRP, Charon and HyCharts.

Other points worth mentioning about our grammar is as follows:

- For making the transformation procedure more clear, we make some terms like `cont_sf` and `ev` explicit in our grammar. To improve readability, `switch` is defined in our grammar as an infix operator where `sf1 switche sf2` denotes a signal function switching from `sf1` to `sf2` when the event `e` occurs. In this continuation form, event's value is ignored for the definition of `sf2` but as the term `cont_sf` illustrates this value

can be used for parameterizing the new signal function and also choosing between more than one continuations.

- The `let-sig` construct can be used for recursive signal function definitions and instantiating signal functions in more than one place.
- Other primitive signal functions and combinators of FRP can be expressed in terms of the ones in our grammar. `integral` can be also defined by using `delay` but since its usage is common in the hybrid systems we include it in our grammar. All arrow combinators can be defined by using `arr`, `first`, and `seq`.
- `edge` is a so-called event source. It takes a boolean signal and produces an event signal where the event occurs each time the boolean signal changes from `False` to `True`. It has the dummy unit value `()`.
- The patterns defined by the *patr* term can be extended to include more of the base language patterns but for a more clear presentation we restrict them only to tuples.

5.4 HS-FRP and RT-FRP

RT-FRP is developed for describing FRP programs for real-time reactive systems where the program can be executed with guaranteed resource bounds. To achieve this end, its type system puts some constraints on the two kind of recursions that can arise in FRP programs. Like our grammar, RT-FRP focuses on the reactive part of an FRP program and assumes certain expressions will be evaluated outside, in the context of a base language.

RT-FRP defines signals instead of signal functions as first class but it is easy to show that for any signal *s* expressable in the RT-FRP grammar, we can come up with a signal function in our grammar where the output signal is *s*. There is no separate event syntactic category in RT-FRP but since events are isomorphic to signals with a `Maybe` type this is not a real distinction.

Although grammars of both languages are equivalent in their power in terms of expressing reactivity, the typing constraints of RT-FRP result in some differences. First of all, the base language of RT-FRP may allow events with higher order values. So such RT-FRP programs are not part of HS-FRP.

On the other hand, to restrict some programs including recursive switches where they can grow arbitrarily large during execution and thus using unbounded space, RT-FRP disallows programs like below (taken from [46] page 45):

$$\begin{array}{l} \text{let-sig } sf = (sf1 \text{ switch}_{ev0} sf) \text{ switch}_{ev1} sf \\ \text{in } sf2 \text{ switch}_{ev2} sf \end{array}$$

When $ev2$ occurs this term becomes:

$$\begin{array}{l} \text{let-sig } sf = (sf1 \text{ switch}_{ev0} sf) \text{ switch}_{ev1} sf \\ \text{in } (sf1 \text{ switch}_{ev0} sf) \text{ switch}_{ev1} sf \end{array}$$

Now suppose $ev0$ occurs, but $ev1$ does not:

$$\begin{array}{l} \text{let-sig } sf = (sf1 \text{ switch}_{ev0} sf) \text{ switch}_{ev1} sf \\ \text{in } ((sf1 \text{ switch}_{ev0} sf) \text{ switch}_{ev1} sf) \text{ switch}_{ev1} sf \end{array}$$

So after everytime $ev0$ occurs without $ev1$ occurring the size of the program becomes larger and because of this, type system of RT-FRP do not allow such expressions.

However, in HS-FRP we are not concerned about resource boundness so we can allow these kinds of recursive switching. How we can translate these kind of FRP programs into hybrid system specifications will be given formally in the next section but intuitively what happens when such a switching occurs is that the new active location becomes the initial location of the switched signal function. So in this example both $ev0$ and $ev1$ in fact defines loops in the hybrid automata where the transition occurs from $sf1$ to again $sf1$ (since it is the initial location of the recursively defined sf).

5.5 Translation

For our translation, we will take a HS-FRP term and describe how a faithful hybrid system specification can be found by syntactically transforming this term. By faithful, we mean a hybrid system where there is a transition for each switching term and the state variable set depends on the signal labelings of the signal network. As discussed in the previous section an FRP program is mainly defining an ultimate signal function and any signal function can be represented as a signal network. In the following discussion, we will often employ Haskell's signal network syntax since it is more familiar.

We will first deal with signal functions which do not include any switching term. Clearly such signal functions will correspond to single node hybrid systems. We will define some syntactic transformations which will put the signal function definition into a canonical form to improve the presentation of the transformation procedure. Then, by using this as a base case, we will recursively show how signal functions including switching can be described as multinode hybrid systems.

First let's define our canonical form which we can use to determine the corresponding hybrid system of a non-switching signal function.

Definition 1 *A signal function network, sf-net term, is in canonical form if:*

- *Each input and output signal is only labeled by a variable,*
- *Left side of let expressions is only a variable rather than a pattern,*
- *The signal function of the signal mapping components (i.e sig-map terms) is not a seq or first term².*

To find out the canonical form of a sf-net term, we apply a series of syntactic transformations which preserve the semantics of the term. Throughout the following transformations, we assume the ability to produce new fresh variables

²Sometimes we will call these signal functions as intermediate signal functions of the network

which are distinct from the ones already used. When we say replacing a variable a with b we mean replacing all the occurrences of a with b . t is a reserved variable representing time, therefore before the transformations we replace t with a new variable if it occurs in the sf-net term.

- First we use the transformation tr_{expand} to expand the intermediate signal functions which are themselves defined as a sf-net. For instance:

```
let-sig sf0 = proc inp -> do
    a <- arr func1 -< inp
    b <- arr func2 -< (a, inp)
    returnA -< a * b
in
    proc inp -> do
        a <- arr func3 -< inp
        b <- sf0 -< a2
        c <- arr func4 -< b
        returnA -< a + b + c
```

is equivalent to:

```
proc inp -> do
    a <- arr func3 -< inp
    let inpsf0 = a2
    asf0 <- arr func1 -< inpsf0
    bsf0 <- arr func2 -< (asf0, inpsf0)
    let b = asf0 * bsf0
    c <- arr func4 -< b
    returnA -< a + b + c
```

Let a signal function, sf , is defined as a sf-net term:

```
sf = proc patsf -> do
    pat1 <- sf1 -< exp1
    ...
    patn <- sfn -< expn
    returnA -< expsf
```

Then if this signal function is used as an intermediate signal function in another sf-net term, we can apply tr_{expand} to replace it with new let expressions and signal mappings which reflect what happens to the signals processed by this signal function:

```
|  |
| --- |
| $tr_{expand}(pat \leftarrow sf \rightarrow exp) =$ |
| $let\ pat_{sf} = exp$ |
| $tr_{expand}(pat_1 \leftarrow sf_1 \rightarrow exp_1)$ |
| ... |
| $tr_{expand}(pat_n \leftarrow sf_n \rightarrow exp_n)$ |
| $let\ pat = exp_{sf}$ |

```

If there is a clash between the variables of sf and the outer signal function's variables as in our example, sf 's such variables should be renamed with fresh variables.

- We use the transformation tr_{comb} on each signal function defined as a seq or first to eliminate them:

```
|  |
| --- |
| $tr_{comb}(pat_i \leftarrow seq\ sf_1\ sf_2 \rightarrow exp_i) =$ |
| $tr_{comb}(v_i \leftarrow sf_1 \rightarrow exp_i)$ |
| $tr_{comb}(pat_i \leftarrow sf_2 \rightarrow v_i)$ |
| $tr_{comb}(pat_i \leftarrow first\ sf \rightarrow exp_i) =$ |
| $tr_{comb}(v_i \leftarrow sf \rightarrow fst\ exp_i)$ |
| $let\ pat_i = (v_i, snd\ exp_i)$ |

```

where v_i is a new variable and fst and snd are the selectors for the first part of a pair and second part of a pair respectively. Being consistent with definitions of seq and $first$, these transformations do not change the semantics of the signal mapping.

- The transformations tr_{exp} and tr_{pat} can be used to eliminate the input and output signal definitions which are not a single variable. Simply replace each exp_i and pat_i with a fresh new variable v_i and add other appropriate let expressions. For expressions, we can simply add $let\ v_i = exp_i$ and use v_i in the place of exp_i . For patterns, we have to add a let expression for each variable bound in the pattern. So for a pattern like $(a, (b, c))$ we can replace the pattern with v_i and add let expressions: $let\ a = fst(v_i)$, $let\ b = fst(snd(v_i))$, and $let\ c = snd(snd(v_i))$.

The transformation $tr = tr_{expand} \circ tr_{pat} \circ tr_{exp} \circ tr_{comb}$ which is applied to each intermediate signal function preserves the semantics of the overall

signal function definition and after the transformation we have an equivalent canonical form of the signal function definition as:

```

proc inp -> do
  let   $x_1 = exp_1$ 
  ...
  let   $x_n = exp_n$ 
   $b_1 \leftarrow sf_1 \prec a_1$ 
  ...
   $b_n \leftarrow sf_n \prec a_n$ 
  returnA  $\prec out$ 

```

We are now ready to define how we can translate a non-switching signal function into a hybrid system through its canonical form.

Definition 2 (Pure Signal Functions) *We call a sf-net term a pure signal function if its syntactic definition does not include:*

- *recursive calls to itself,*
- *recursively defined signal functions,*
- *signal functions defined by switch*

Since there is no switching happening in a pure signal function there will be no discrete state changes at the computation of state variables and the corresponding hybrid system will have only one vertex v . If we assume that the canonical form of the signal function in question is as above we define the corresponding hybrid system as having only one location, $V = \{v\}$, thus no transitions and the variable set: $X = \{inp, x_1, \dots, x_n, b_1, \dots, b_n, out\} \cup X_e$.

Here X_e is a special set of variables whose definitions are based on the possible event signals that are part of the signal network. Later when we introduced switching these variables will be helpful.

In the canonical form of a pure signal function, each intermediate signal function is defined by one of the following: localTime, arr, const, delay, edge, or

integral. We have to give an evolution function for each of these signal mappings where the function is consistent with how the input signal is converted to an output signal.

Before starting to define the evolution functions, we have to first replace all variables b_i in the canonical form by the reserved variable t if they are defined as:

$$b_i \leftarrow \text{localTime} \rightarrow a_i$$

since `localTime` extracts the time information from a signal (i.e. for each sample value of a_i returns the sampling time). We have to also remove b_i from X .

The evolution functions of variables in X can be defined by using the canonical form:

- For each let $x_i = \text{exp}_i$ in the canonical form:

We can assume that x_i is a function of the variables $\{y_1, y_2, \dots, y_n\}$ that appear in exp_i . Moreover let's assume that the first m of these variables are HS-FRP variables (i.e signal labels) and the rest are part of the base language. Then the corresponding evolution function will be:

$$\varphi_v(x_i)(t) = f(y_1(t), y_2(t), \dots, y_m(t), y_{m+1}, y_{m+2}, \dots, y_n)$$

For instance the evolution function corresponding to :

$$\text{let } a = k * \text{sqrt}(b + c + d)$$

would be:

$$\varphi_v(a)(t) = k * \text{sqrt}(b(t) + c + d(t))$$

assuming b and d are signal labels.

- For each intermediate signal function (or *sig-map* term of the grammar) in the canonical form depending on the signal function we define the evolution functions as:

$b_i \leftarrow \text{arr } f \text{ } \prec a_i$	$\varphi_v(b_i)(t) = f(a_i(t))$
$b_i \leftarrow \text{const } b_0 \text{ } \prec a_i$	$\varphi_v(b_i)(t) = b_0 \forall t$
$b_i \leftarrow \text{delay } b_0 \text{ } \prec a_i$	$\varphi_v(b_i)(t) = a_i(t_{prev})$ $\varphi_v(b_i)(0) = b_0$
$b_i \leftarrow \text{integral } \prec a_i$	$\varphi_v(b_i)(t) = \int a(t)dt$

Note that `arr` lifts a function and applies it pointwise to the input signal, `const` produces a signal function with a constant output signal value, and `delay` delays the input signal by one time step. Another possibility is that `sf` is the event source edge. How to handle this case is explained below.

- Finally, we construct the set X_e by adding two variables e_{val} and e_{pred} for each variable e denoting an event signal. e_{val} carries the value of the event which may be useful for switching. e_{pred} is a boolean variable which becomes True when the event occurs. The evolution functions for these variables are defined as below where $e(t)$ is the current event value:

$$\begin{aligned} \varphi_v(e_{pred})(t) &= \begin{cases} \text{True} & e(t) = \text{Event } c \\ \text{False} & e(t) = \text{NoEvent} \end{cases} \\ \varphi_v(e_{val})(t) &= \begin{cases} c & e(t) = \text{Event } c \\ \text{undefined} & e(t) = \text{NoEvent} \end{cases} \end{aligned}$$

Let e be a label of the output signal of the primitive event source edge and b be the boolean input signal. Here is how the values of e over time is computed which then can be used to get e_{val} and e_{pred} as described above:

$$e \leftarrow \text{edge } \prec b$$

$$e(t) = \begin{cases} \text{Event } () & b(t) = \text{True} \ \&\& \\ & b(t_{prev}) = \text{False} \\ \text{NoEvent} & \text{otherwise} \end{cases}$$

The functionality of the event operators `mergeE` and `tagE` can be easily defined in terms of the newly introduced e_{val} and e_{pred} variables. For instance, `mergeE` produces an event from two existing ones, where the new event occurs every time either occurs, with the value of that event at

that time. If both of the events occur at the same time the first event's value has priority:

$$\begin{aligned} e &= \text{mergeE } e1 \ e2 \\ e_{pred} &= e1_{pred} \parallel e2_{pred} \\ e_{val}(t) &= \begin{cases} e1_{val}(t) & e1_{pred} \\ e2_{val}(t) & \text{otherwise} \end{cases} \end{aligned}$$

Another commonly used event operator is `tagE` which changes an event's value:

$$\begin{aligned} e &= \text{tagE } e1 \ exp \\ e_{pred} &= e1_{pred} \\ e_{val}(t) &= \begin{cases} exp & e_{pred} \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

- We do not give an evolution function for the variable `inp` since it is assumed that its values will come from the environment.

This completes the first part of our translation. Now let's consider the cases where switching is involved in the definition of a signal function.

Definition 3 *Let $K\text{-SF}$ denote the set of sf-net terms which are:*

- *either a pure signal function,*
- *or a signal function defined by switch provided that the signal function arguments of the switch operator are also in $K\text{-SF}$. Let sf_{left} and sf_{right} denote the initial and continuation signal functions respectively.*

We will use an induction based recursive translation for the $K\text{-SF}$ signal functions. Our induction is based on the syntax tree of a signal function, sf , in $K\text{-SF}$. First, we assume that if sf is defined by switch then:

- the definition of sf does not include any recursively defined signal functions or itself,
- the continuation is not a selection (i.e. not the third kind of continuation involving if).

We will show later how to deal with the cases without these assumptions. If these assumptions hold for sf , then we can recursively define its FRP syntax tree as:

- If sf is a pure signal function then it is a node (leaf) labeled by sf .
- If sf is defined by switch then the FRP syntax tree has the root labeled as $switch_{sf}$, the left subtree of root is defined by sf_{left} and the right subtree of root is defined by sf_{right} .

Let $(V, X, \Rightarrow) \leftarrow sf$ mean that $(V, X, \Rightarrow)$ is the hybrid system corresponding to signal function sf . We omit the evolution functions from the hybrid system specification since switching does not change the evolution functions of state variables.

Our induction is based on the depth of the FRP syntax tree of sf . The base case will be a tree with depth 0 which means it is a pure signal function. Then by using the translation given for pure signal functions we can find a hybrid system: $(\{v\}, X_1, \emptyset) \leftarrow sf$

Our induction hypothesis is that for a signal function sf in K-SF with an FRP syntax tree of depth n or lower we know that: $(V, X, \Rightarrow) \leftarrow sf$

Then the corresponding hybrid system for a signal function with an FRP syntax tree of depth $n + 1$ is defined as below:

- We know by the induction hypothesis that:

$$\begin{aligned} (V_1, X_1, \Rightarrow_1) &\leftarrow sf_{left} \\ (V_2, X_2, \Rightarrow_2) &\leftarrow sf_{right} \end{aligned}$$

since the depth of the FRP syntax trees of sf_{left} and sf_{right} is n . Let b_1 and b_2 be the final variables of sf_{left} and sf_{right} respectively, e represent the event used for switching and c be the lambda variable that may be used to parameterize sf_{right} .

- $V = V_1 \cup V_2$

- $X = X_1 \cup X_2$

We have to synchronize the names of the final variables of sf_{left} and sf_{right} if they are not the same since they represent the same output. So if they are different, we simply rename b_2 as b_1 and if b_1 is used already as a variable in sf_{right} , we rename that one too with a new variable.

- We can assume that the hybrid systems of sf_{left} and sf_{right} are represented as super states and there is an edge between them defined by e_{pred} . Formally the edges of the new hybrid system is defined as:
 $\Rightarrow = \Rightarrow_1 \cup \Rightarrow_2 \cup \{v_i \in V_1 | (v_i, e_{pred}, F(c) = e_{val}, v_2)\}$
 where v_2 is the initial location of V_2 .

This means when the event occurs the control passes to the initial location of sf_{right} regardless of which location was active in sf_{left} and the lambda variable c is initialized by event e 's final value.

- $(V, X, \Rightarrow) \leftrightarrow sf$

Next let's consider the case where the event's value is used as a selector between more than one possible next signal functions by an if expression. Let $sf1$ be the initial signal function, $sf2$ be the continuation signal function when ext_pred is true and $sf3$ be the one when it is false. Then the recursive step of the translation for this case is:

- We know that:

$$\begin{aligned} (V_1, X_1, \Rightarrow_1) &\leftrightarrow sf1 \\ (V_2, X_2, \Rightarrow_2) &\leftrightarrow sf2 \\ (V_3, X_3, \Rightarrow_3) &\leftrightarrow sf3 \end{aligned}$$

from the induction hypothesis.

- $V = V_1 \cup V_2 \cup V_3$
- $X = X_1 \cup X_2 \cup X_3$ where final variables of X_2 and X_3 are synchronized by X_1 's as above.

- $\Rightarrow = \Rightarrow_1 \cup \Rightarrow_2 \cup \Rightarrow_3 \cup E_1 \cup E_2$
 $E_1 = \{v_i \in V_1 | (v_i, e_{pred} \wedge ext_pred, F(c) = e_{val}, v_2)\}$
 $E_2 = \{v_i \in V_1 | (v_i, e_{pred} \wedge !ext_pred, F(c) = e_{val}, v_3)\}$ where v_2 is the initial location of V_2 and v_3 is the initial location of V_3
- $(V, X, \Rightarrow) \leftarrow sf$

Finally let's define how we can construct the hybrid system when sf contains recursive definitions. A recursive call to a signal function sf_i means that the system moves to the initial location of sf_i , that is a loop in the hybrid system. To deal with this case we represent every recursive call to sf_i as a leaf node in the FRP syntax tree labeled as $loop_{sf_i}$ and define a dummy one vertex hybrid system corresponding to this node as: $(\{loop_{sf_i}\}, \emptyset, \emptyset) \leftarrow sf_i$

Then after following the above steps with considering this definition we will find a translation: $(V, X, \Rightarrow) \leftarrow sf$.

We then follow the following steps and transform the hybrid system to define the loop in the system:

For each vertex $loop_{sf_i}$ in V do:

- Replace each edge $(v_i, B(X), F(X), loop_{sf_i})$ with $(v_i, B(X), F(X), v_1)$ where v_1 is the initial location of V .
- Delete each edge $(loop_{sf_i}, B(X), F(X), v_i)$. We delete each outgoing edge from $loop_{sf_i}$ since they will be never taken. This can make some part of the hybrid system inaccessible but this is indeed a true fact. For instance, in:

let-sig $sf=sf1$ switch ($sf2$ switch (sf switch $sf3$))) in sf

the system will never switched to $sf3$ since every time it switches to sf from $sf2$ what really happens is a switch to the initial location $sf1$.

- Finally we delete the vertex $loop_{sf_i}$ from V .

An interesting case is what happens when $v_1 = loop_{sf_i}$ which means the leftmost signal function in the possibly nested switch declarations of sf_i is a call to sf_i . Then we end up with the dummy state as the initial location with no outgoing edges. This hybrid system will probably represent the situation reasonably since such a declaration of sf_i will halt the system because there will be an infinite computation.

For allowing intermediate signal functions defined by switch in pure signal functions, we can first transform the signal function into one in K-SF by cloning it and then apply our induction. For instance:

```
sf = proc inp -> do
  ...
  b1 <- sf1 switche lambda.c sf2 -< a1
  b2 <- sf3 switche lambda.d sf4 -< a2
  ...
  returnA -< out
```

We can define two signal functions based on sf where the only difference is at the switching signal mappings:

```
sfinit = proc inp -> do
  ...
  b1 <- sf1 -< a1
  b2 <- sf3 -< a2
  ...
  returnA -< (out,e)
```

```
sfnext = proc inp -> do
  ...
  b1 <- sf2 -< a1
  b2 <- sf4 -< a2
  ...
  returnA -< out
```

```
sf = sfinit switche lambda.c sfnext
```

Note that we also have to rename the variable d in the definition of sf_4 to c .

When more than one event is used in a number of switch expressions things become more tricky. For instance, if there are three events labeled by e_1, e_2

and e_3 then we have to have 8 clones, sf_{000} , sf_{001} , sf_{010} , sf_{011} , sf_{100} , sf_{101} , sf_{110} , sf_{111} . The signal function $sf_{d_1 d_2 d_3}$ includes the initial signal functions of switch expressions defined by e_i when d_i is 0 and the switched to signal functions when d_i is 1. So we have to add a switch expression with the event e_i from $sf_{d_1 d_2 d_3}$ to $sf_{d'_1 d'_2 d'_3}$ if d_i is 0 and d'_i is 1.

5.5.1 Bug Algorithm as a Hybrid System

Let us now give an example FRP program and provide the corresponding hybrid system to illustrate how this translation works. The Bug navigation algorithm is proposed in [28] where the robot drives towards a goal. When an obstacle is encountered, the robot circles the obstacle, finds the closest point to the goal while circling and then returns to this point and continues driving to the goal. Figure 5.7 gives the HS-FRP coding of this algorithm.

It is assumed that the input stream coming from the robot will include the position of the robot, the sonar values and a boolean flag denoting if the robot is blocked. The base language calls `position`, `sonars`, and `isBlocked` returns the relevant information for any given input signal value.

The top-level signal function `bug` is defined by the pure signal functions `goto_goal`, `circle_once`, `circle_to`, and `done` so there should be four locations in the overall hybrid system. Each of these pure signal functions output an imaginary force vector which is used to drive the robot. In the first one the force vector is computed from robot's position and the goal's position, for the circling tasks it is computed from sonar values by a wall following algorithm. So `wallFollow` is a call to the base language that computes and returns the vector value for the current values of the sonars.

The first signal function `goto_goal` terminates either when the robot is blocked or robot's position equals to goal's. The event value is the position of the robot when the termination occurs. Depending on this event value the next mode may be `done` which means the task is accomplished or `circle_once` if an obstacle is encountered.

```

let-sig bug =
  let-sig goto_goal =
    proc inp -> do
      let vec = driveTo goal pos
      pos      <- arr position  -< inp
      blocked <- edge           -< isBlocked inp
      done     <- edge           -< (goal==pos)
      returnA -< (vec, tagE (mergeE blocked done) pos)
    in
  let circle_once =
    proc inp -> do
      let vec = wallFollow sonVals
      let p   = closestPoint goal pos p_prev
      pos     <- arr position  -< inp
      sonVals <- arr sonars    -< inp
      p_prev  <- delay maxInt -< p
      done    <- edge          -< (pos0==pos)
      returnA -< (vec, tagE done p)
    in
  let circle_to =
    proc inp -> do
      let vec = wallFollow sonVals
      pos     <- arr position  -< inp
      sonVals <- arr sonars    -< inp
      done    <- edge          -< (p0==pos)
      returnA -< (vec, done)
    in
  let done =
    proc inp -> do
      vec <- const zeroVec -< inp
      returnA -< vec
    in

  goto_goal switch
  (lambda.pos0 if pos0==goal then done
    else (circle_once switch
      (lambda.p0 circle_to switch
        bug
        )))

```

Figure 5.7: Bug Navigation Algorithm

In `circle_once` the robot circles around the obstacle and the termination criteria checks if the robot returned to the initial position where it started circling. Similarly in `circle_to` the robot circles around the obstacle to get to the closest point which is computed during `circle_once`. The signal function terminates when it arrives to this position. Then in the definition of `bug` there is a recursive call at this point which means the next location after `circle_to` is the initial location `goto_goal`.

Now let's look at how these definitions can be transformed into a formal hybrid system definition. The top level signal function `bug` is defined by a series of `switch` operators where all the three kind of continuation styles (i.e. *cont_sf* terms of the grammar) are employed and there is also a recursive call. The initial signal function is `goto_goal`. Since this is a pure signal function first we translate it to a single location. The canonical form of `goto_goal` is:

```
let-sig bug =
  let-sig goto_goal =
    proc inp -> do
      let vec = driveTo goal pos
      let v_1 = isBlocked inp
      let v_2 = goal==pos
      pos      <- arr position  -< inp
      blocked <- edge           -< v_1
      done    <- edge           -< v_2
      returnA -< (vec, tagE (mergeE blocked done) pos)
```

where `v_1` and `v_2` are new variables. Then we can determine the evaluation functions as:

$$\begin{aligned}\varphi_{goto_goal}(vec)(t) &= \text{driveTo } goal \ pos(t) \\ \varphi_{goto_goal}(v_1)(t) &= \text{isBlocked } inp(t) \\ \varphi_{goto_goal}(v_2)(t) &= goal == pos(t) \\ \varphi_{goto_goal}(pos)(t) &= \text{position } inp(t)\end{aligned}$$

Note that since the variable `goal` does not represent a signal in the network, it is not time dependent (i.e its value remains same for each location and time instant) and it is an argument of the hybrid system. For HS-FRP this means its value will be determined by a call to the base language.

For each event variable in the network there will be two variables in the hybrid system. The termination event is denoted by e :

$$\begin{aligned}
\varphi_{goto_goal}(blocked_{val})(t) &= \begin{cases} () & v_1(t) = True \ \&\& \\ & v_1(t_{prev}) = False \\ undefined & otherwise \end{cases} \\
\varphi_{goto_goal}(blocked_{pred})(t) &= \begin{cases} True & v_1(t) = True \ \&\& \\ & v_1(t_{prev}) = False \\ False & otherwise \end{cases} \\
\varphi_{goto_goal}(done_{val})(t) &= \begin{cases} () & v_2(t) = True \ \&\& \\ & v_2(t_{prev}) = False \\ undefined & otherwise \end{cases} \\
\varphi_{goto_goal}(done_{pred})(t) &= \begin{cases} True & v_2(t) = True \ \&\& \\ & v_2(t_{prev}) = False \\ False & otherwise \end{cases} \\
\varphi_{goto_goal}(e_{pred})(t) &= blocked_{pred} \ || \ done_{pred} \\
\varphi_{goto_goal}(e_{val})(t) &= \begin{cases} pos(t) & e_{pred} \\ undefined & otherwise \end{cases}
\end{aligned}$$

This seems complicated but it means that the event's value is the value of the state variable pos at the time the event predicate becomes true. The event predicate becomes true when either `isBlocked` begins returning true ($blocked_{pred}$) or the value of pos equals to $goal$ ($done_{pred}$). The evolution functions for other pure signal functions and related events can be translated similarly.

Now let's look at how the transitions are defined based on the switch operators. In the first switch, where the initial signal function is `goto_goal`, the continuation is a conditional. We will have two outgoing transitions from `goto_goal` for each branch. Because of the lambda expression, both transitions will initialize the variable $pos0$ to the event value e_{val} . For the first transition, the predicate is that both e_{pred} and $pos0 == goal$ are true. The next location is the location defined by the pure signal function `done`. For the else case the predicate is e_{pred} is true but $pos0 == goal$ is false. This time the next location is also defined by switch. So the transition will be from `goto_goal` to the initial location of this switch term which is `circle_once`.

The transition from `circle_once` is not a conditional which means the predicate is simply e_{pred} and the next location is `circle_to` again the first location of a switch term. However, there is still an information flow between the locations because of the lambda type continuation. The variable $p0$ of `circle_to` is initialized as the event value e_{val} . This event value comes from `circle_once`'s state variable p (see the tag expression) which computes the closest point to the goal around the obstacle. `circle_to` will drive the robot to this point and then resume driving to the goal.

The final switch defines the last transition of the hybrid system. The initial location is `circle_to` and since it is the simplest continuation type, the predicate is e_{pred} and there is no initialization of a variable. The continuation signal function is `bug`. Since this is a recursive call to the top level signal function, the transition will be from `circle_to` to the initial location of `bug` which is `goto_goal`.

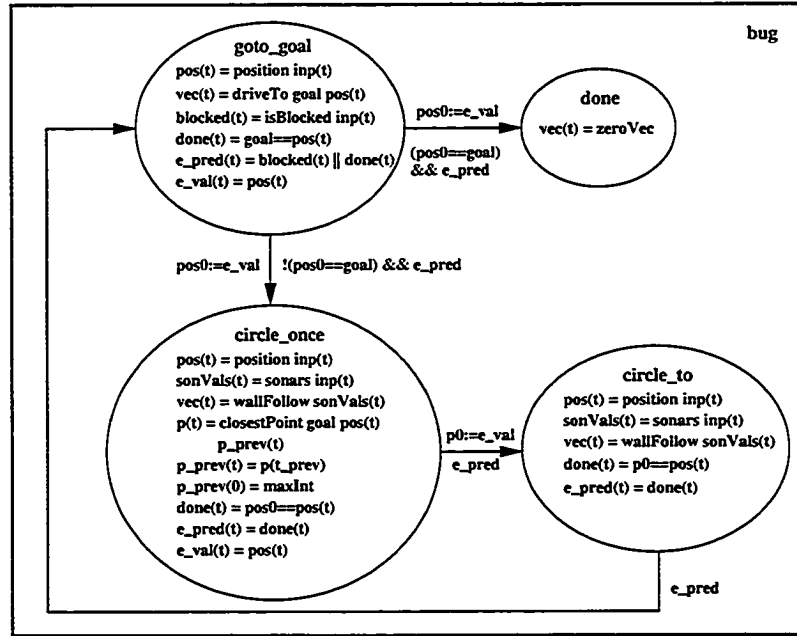


Figure 5.8: The hybrid system for the bug algorithm

Figure 5.8 shows the hybrid system for the bug algorithm defined as a HS-FRP program. The fresh variables generated during the translation (like

v_1 and v_2 of `goto_goal`) are replaced in the equations with their respective evolution functions for a better representation. For the same reason, the evolution functions of event variables are simplified.

Chapter 6

Conclusions

To date, robot programming has been an area that has been 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.

In this thesis, we present a domain-specific language for robot programming developed by using the FRP hybrid systems framework. We demonstrated the capabilities of the FRP-based language, Frob, through a comparative review of other languages and building a tracking-based robot navigation system. We show that 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.

The strengths of Frob emerged from our work can be summarized 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 separating 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 programming. So one can use whatever architecture best suits the problem in question. For example using both the FSM depended task definitions or subsumption architecture is possible in Frob. Since developing abstractions is relatively 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. A generic robot control algorithm for a specific problem can be used for different type of robots after importing the required sensory and actuator methods specific to the robot into from via a clean interface provided.
- FRP's rich set of combinators and operators to define events and behaviors combined with the task abstraction and task transformation operators makes Frob unique among other robot languages in the sense that Frob programs can deal in detail with both the reactive and planning aspects of the domain.
- By using the same FRP framework other domains like vision or user interfaces related to robot programming can be integrated into Frob with a little effort.

In this thesis we also have taken the first steps toward establishing an explicit, syntactic connection between FRP and hybrid systems. Since hybrid systems are Turing complete, their expressive power is, in principle, equivalent. However, we believe that relating the two to each other at a syntactic level will result in a better understanding and formal analysis of FRP programs. We show how a formally specified hybrid system can be expressed in FRP and how a corresponding hybrid system specification can be found for a given FRP program. This makes the algorithmic formal analysis methods available for FRP and the domain specific languages developed in it. This may also lead to techniques that compare Frob with other robot programming systems if similar reductions to hybrid systems can be found for them. Another longer-term goal may be creating analysis tools for complex software systems for robotics.

Appendix A

A Translator from FRP to Hybrid System Specifications

```
-----  
--  
-- This module defines the abstract data types for the parsed FRP  
-- programs and the formal hybrid system specification.  
--  
-----  
  
module TranslatorTypes where  
  
-- Main data type for signal functions. SigFn is for signal networks  
-- and the rest denote the primitive signal functions.  
  
data SigFn = SigFn (Name,Patr,Expr,[SigMap])      |  
             Switch (SigFn,Maybe Patr,SigFn)      |  
             Switch2 (SigFn,Patr,Pred,SigFn,SigFn) |  
             Arr Expr                               |  
             Seq (SigFn,SigFn)                     |  
             Fst SigFn                             |  
             LocalTime                             |  
             Const Expr                             |  
             Delay Expr                             |  
             Edge                                    |  
  
-- SigMap denotes signal mappings (i.e intermediate signal functions  
-- in a signal network)
```

```

type SigMap = (Patr,SigFn,Expr)

-- LetE denotes let expressions.

type LetE   = (String,String)

--- Type for events.

data Ev     = MergeE (Expr,Expr) |
              TagE (Expr,Expr)

-- Auxilary types used in SigFn.

type Name = String
data Patr = Var_p Name | Pair (Patr,Patr)
data Expr = Var_e Name | Expr String | Expr_e Ev |
              Expr_f (Name, Expr) | Var_f Name | Expr_p (Expr,Expr)

-- Main data type for hybrid systems.

type HS     = (Name,[Loc],[Trans],[Name,[EvFn]])

-- Trans denotes transitions between locations.

type Trans = (Loc,Pred,Loc,[EvFn])

-- LocS denotes single state hybrid systems and LocH is used for
-- denoting meta states

data Loc   = LocS (Name,StVar,[EvFn]) | LocH HS

-- Predicates and state variables are simply strings.

data Var   = Var Name
data Pred  = Pred Name

-- EvFn denotes evolution functions.

data EvFn = EvFn_C (Name,Expr) | EvFn_D (Name,Expr,Expr) |
              EvFn_E {var_e::Name, when::Expr, val::Expr, isEdge::Bool}

-- StVar denotes state variables. As a difference from the formal
-- definitions they are categorized.

```

```

data StVar = StVar {inp::[Name], out::[Name], ev::Maybe Name,
                    local::[Name]}

-- A bunch of instance declarations for printing and equivalency check
-- of the new data types.

instance Eq Pred where
  (==) (Pred str1) (Pred str2) = str1==str2

instance Eq EvFn where
  (==) ef1 ef2 =
    case (ef1,ef2) of
      (EvFn_C (n1,_),EvFn_C (n2,_)) -> n1 == n2
      (EvFn_D (n1,_,_),EvFn_D (n2,_,_)) -> n1 == n2
      (EvFn_E {var_e=n1},EvFn_E {var_e=n2}) -> n1 == n2
      _ -> False

instance Eq Loc where
  (==) (LocS (n1,vars1,ef1)) (LocS (n2,vars2,ef2)) =
    (n1==n2) && (vars1==vars2) && (ef1==ef2)
  (==) _ _ = False

instance Eq StVar where
  (==) sv1 sv2 =
    let StVar {inp=inp1, out=out1, ev=e1, local=loc1} = sv1
        StVar {inp=inp2, out=out2, ev=e2, local=loc2} = sv2
    in
      (inp1==inp2) && (out1==out2) && (e1==e2) && (loc1==loc2)

instance Show EvFn where
  show = print_fs

print_fs (EvFn_C (o,e))=o++"="++pr_exp e
print_fs (EvFn_D (o,e0,e))=o++"=("++pr_exp e0++") "++pr_exp e
print_fs (EvFn_E {var_e=o, when=e1, val=e2})=o++"="++pr_exp e1++"... "
                                                ++pr_exp e2

instance Show Expr where
  show = pr_exp

pr_exp :: Expr -> String
pr_exp e = case e of
  Var_e n -> n
  Expr str -> str

```

```

Expr_e (MergeE (e1,e2)) -> "("++pr_exp e1++") 'merge'
                               ("++ pr_exp e2++)"
Expr_e (TagE (e1,e2)) -> "("++pr_exp e1++") 'tag' "
                               ++pr_exp e2
Expr_f (n,e') -> n++ "("++(pr_exp e')++)"
Var_f n -> n
Expr_p (e1,e2) -> "("++pr_exp e1++","++pr_exp e2++)" "

```

```

-----
---
--- This module provides a translator from a FRP signal function
--- definition to the hybrid system specification. It is also
--- possible to output the hybrid system specification as a dot
--- graph language file in order to display it as a graph.
---
-----

module Main where

import TranslatorTypes
import System (getArgs)
import List ((\\), nub)
import Char

-- This is the top-level function for the translation.

transform :: SigFn -> Name -> HS
transform sf name= case sf of

-- First case: Pure Signal Functions. Translates to a single location.

SigFn(n,inp,out,ls) ->
  let sf'=replace_t sf -- eliminate localTime
      (sfs, lets0) = canonize sf' -- put into canonical form
      lets = expand_lets lets0 -- condense let expressions
      -- tr converts signal mappings to evolution functions
      tr :: [SigMap] -> [EvFn]
      tr [] = []
      tr ((Var_p o,l,Var_e i):sfs) = (case l of
        Arr (Var_f f) -> [EvFn_C (o,Expr_f (f,Var_e i))]
        Arr _ -> error "lifted function as an expression"
        Const(Expr str) -> [EvFn_C (o,Expr str)]
        Delay(Expr str) -> [EvFn_D (o,Expr str, Var_e i)]
        LocalTime -> error "localtime should be eliminated"
        sf -> error ("tr : unexpected SF")
      )++tr sfs
      inps = expand_inp inp -- find input variables
      (outs,e,_) = expand_out out True 1 -- find output variables and
      outs_e = case e of -- the event if existing
        Nothing -> []

```

```

        Just ef -> [ef]
outs' = filter (\f -> case f of
                        EvFn_C (n, Expr "") -> False
                        _ -> True )    (outs++outs_e)
fs0 = expand_let lets ((tr sfs)++outs')
fs = add_t (inps++(concat $
                    map (\e-> [e++"_val",e++"_pred"])) xe)) fs0
x = getVars fs
xe = geteVars fs0
x_out = (nub $ getVars outs) \\ (nub $ getVars outs_e)
x_loc = (nub $ getVars fs) \\ (nub x_out)
e_name = case e of
    Nothing -> Nothing
    Just e_f -> Just (head (getVars [e_f]))
vars = StVar {inp=inps, out=x_out, ev=e_name, local=x_loc}
loc = LocS (n,vars,fs)
in
(n, [loc], [], (n,[]))

```

-- Second case : Switching to a single signal function

```

Switch(sf1,patr,sf2) ->
let (n1, locs1, arcs1, (n01, inits1)) = transform sf1 "init"
    (n2, locs2, arcs2, (n02, inits2)) = transform sf2 "next"
    LocS (_,vars1,ef1) = head locs1
    inp1 = inp vars1
    out1 = out vars1
    ev1 = case ev vars1 of
        Nothing -> error "Switch : no event"
        Just n -> n
    LocS (_,vars2,ef2) = head locs2
    inp2 = inp vars2
    out2 = out vars2
    init=case patr of
        Nothing -> []
        Just (Var_p n) -> [EvFn_C (n,Expr (ev1++"_val"))]
        Just _ -> error "multiple output"
-- name syhncronization
nsync ef2 =
    let repE exp = foldl (\exp (a,b) -> replace_E [b] a exp)
                        exp (zip (inp1++out1) (inp2++out2))
        repS n    = foldl (\str (a,b) -> replace [b] a str)
                        n (zip (inp1++out1) (inp2++out2))

```

```

in case ef2 of
  EvFn_C (n,exp) -> EvFn_C (repS n, repE exp)
  EvFn_D (n,exp1,exp2) -> EvFn_D (repS n, repE exp1,
                                   repE exp2)
  EvFn_E {var_e=n, when=e1, val=e2, isEdge=b} ->
    EvFn_E {var_e=repS n, when=repE e1, val=repE e2,
            isEdge=b}
locs2'=map (\(LocS (n2,vars2,ef2)) ->
            LocS (n2,vars2 {inp=inp1, out=out1}, map nsync ef2))
  locs2
locs=locs1++locs2'
arc = map (\l -> let LocS (_,vars,_) = l
                Just n = ev vars
                in (l,Pred (n++"_pred"), head locs2', init))
  locs1
arcs=arcs1++arcs2++arc
in (name, locs, arcs, (n01, inits1++inits2))

```

-- Third case : Continuation is based on the event value

```

Switch2 (sf1,patr,Pred pred,sf2,sf3) ->
let sf12 = Switch (sf1, Just patr, sf2)
    sf13 = Switch (sf1, Just patr, sf3)
    (n1, locs1, arcs1, (n01, inits1)) = transform sf1 "init"
    (n2, locs2, arcs2, (n02, inits2)) = transform sf2 "next1"
    (n3, locs3, arcs3, (n03, inits3)) = transform sf3 "next2"
    (n12, locs12, arcs12, (_, inits12)) = transform sf12 "fst"
    (n13, locs13, arcs13, (_, inits13)) = transform sf13 "snd"
locs'=(locs12++locs13)
arcs'=(arcs12++arcs13)
g (LocS (n,_,_)) = n
locs1n = map g locs1
locs2n = map g locs2
locs3n = map g locs3
inits'=nub (inits12++inits13)
f arc@(l1@(LocS (n1,_,_)),Pred str,l2@(LocS (n2,_,_)),inits) =
  if elem n1 locs1n
  then if elem n2 locs2n
    then (l1,Pred (str++" && "++pred),l2,inits)
    else if elem n2 locs3n
      then (l1,Pred (str++" && !("++pred++")),l2,inits)
      else arc
  else arc

```

```

        arcs''=map f arcs'
    in
        (name,locs',arcs'',(n01,inits'))

l -> error "Unexpected signal function in transform"

-- Used for replacing the generated new variables in the evolution
-- functions with their let definition in order to improve readability
-- and minimize the state variable set.

expand_let :: [LetE] -> [EvFn] -> [EvFn]
expand_let [] fs = fs
expand_let ((v,str):rest) fs =
    let repl (EvFn_C (o,e)) = EvFn_C (o,(replace_E [v] str e))
        repl (EvFn_D (o,e0,e)) = EvFn_D (o,e0,(replace_E [v] str e))
        repl (EvFn_E {var_e=o, when=e1, val=e2, isEdge=b})=
            EvFn_E {var_e=o, when=replace_E [v] str e1,
                    val=replace_E [v] str e2, isEdge=b}
    in
        fs'=map repl fs
    in
        expand_let rest fs'

-- Filters variables in the evolution functions.

getVars :: [EvFn] -> [Name]
getVars [] = []
getVars (f:rest) = case f of
    EvFn_C (o,_) -> [o]++getVars rest
    EvFn_D (o,_,_) -> [o]++getVars rest
    EvFn_E {var_e=o} -> [o]++getVars rest

-- Filters event variables in the evolution functions.

geteVars :: [EvFn] -> [Name]
geteVars [] = []
geteVars (f:rest) = case f of
    EvFn_E {var_e=o} -> [o]++geteVars rest
    _ -> geteVars rest

-- Marks variables in the evolution functions that are signal
-- labelings as depending on time.
-- Example :  $a(t)=\sqrt{b+c^2+3*d}$  becomes
--            $a(t)=\sqrt{b(t)+c^2+3*d(t)}$  if b and d are signal labels

```

```

add_t :: [Name] -> [EvFn] -> [EvFn]
add_t ns [] = []
add_t ns fs =
  let os = ns ++ getVars fs
      add [] f = case f of
        EvFn_E {var_e=o, when=e1, val=e2, isEdge=b} ->
          let Expr str=e1
              str2 = " && (!"++o++"_pred(t_prev))"
          in if b then
              EvFn_E {var_e=o, when=Expr (str++str2),
                      val=e2, isEdge=b}
            else f
        _ -> f
      add (v:vs) f=
        let v' = v++"(t)"
            v'' = v++"(t_prev)"
        f' = case f of
          EvFn_C (o,e) ->
            EvFn_C (o,(replace_E [v] v' e))
          EvFn_D (o,e0,e) ->
            EvFn_D (o,e0,(replace_E [v] v'' e))
          EvFn_E {var_e=o, when=e1, val=e2, isEdge=b} ->
            let Expr str = replace_E [v] v' e1
            in EvFn_E {var_e=o, when=Expr str,
                      val=replace_E [v] v' e2, isEdge=b}
        in
          case f of
            EvFn_C (o,Expr_e (MergeE(Var_e e1,Var_e e2))) ->
              let whene=Expr $ e1++"_pred || "++e2++"_pred"
                  vale =Expr $ e1++"_val (if "++e1++"_pred"++)\\1"
                      ++o++"_val(t)="+e2++
                      "_val (otherwise)"
              in add vs (EvFn_E o whene vale False)
            EvFn_C (o,Expr_e (TagE(Var_e e1, exp))) ->
              let whene = Expr $ e1++"_pred"
                  vale = exp
              in add vs (EvFn_E o whene vale False)
            _ -> add vs f'
  in
    map (add os) fs

```

-- Puts the signal function in canonical form by applying various

```
-- syntactic transformations on signal mappings. The second argument
-- of expand is used as a counter for generating fresh variables.
```

```
canonize :: SigFn -> ([SigMap],[LetE])
canonize sf = case sf of
  SigFn(n,inp,out,ls) -> let (sfs, lets,_) = expand ls 0
                        in (sfs,lets)
  _ -> error "canonize : wrong format sf"
```

```
expand :: [SigMap] -> Int -> ([SigMap],[LetE],Int)
```

```
expand [] cnt = ([],[],cnt)
```

```
-- Replace patterns with variables
```

```
expand ((Pair (p1,p2), l,i):ls) cnt = case l of
  Fst sf -> let (i1,i2) = case i of
    Expr str -> (Expr ("fst(++str++)"),
                "snd(++str++)")
    Var_e str-> (Expr ("fst(++str++)"),
                "snd(++str++)")
    _ -> error "expand : case1 i"
  lets = case p2 of
    Var_p v -> [(v, i2)]
    p -> parseP p i2
  i' = case i of
    Var_e v -> Expr ("fst(++v++)")
    Expr str -> Expr ("fst(++str++)")
    _ -> error "expand : case1 i"
  (sfs,lets',cnt2) = expand ((p1,sf,i'):ls) cnt
  in (sfs,lets++lets',cnt2)
  _ -> let v = "v_"++intToString cnt
  lets = parseP (Pair (p1,p2)) v
  (sfs,lets',cnt2) = expand ((Var_p v,l,i):ls) (cnt+1)
  in
  (sfs,lets++lets',cnt2)
```

```
-- Replace expressions with variables
```

```
expand ((o,l,Expr str):ls) cnt =
  let v = "v_"++intToString cnt
  lets = [(v,str)]
  (sfs,lets',cnt2) = expand ((o,l,Var_e v):ls) (cnt+1)
  in
  (sfs,lets++lets',cnt2)
```

```

expand ((o,l,Expr_p (e1,e2)):ls) cnt =
  let v = "v_" ++ intToString cnt
      str = ("++show e1++", "++show e2++")
      lets = [(v,str)]
      (sfs,lets',cnt2) = expand ((o,l,Var_e v):ls) (cnt+1)
  in
    (sfs,lets++lets',cnt2)

expand ((Var_p o,l,Var_e i):ls) cnt = case l of
  -- Expand signal mappings that are also signal networks
  SigFn(n,Var_p inp,Var_e out,lls) ->
    let (lls1,lets1,cnt2)=expand lls cnt
        lls2=map (\(Var_p o',s,Var_e i') ->
          let (new_o,new_i)=(if o'==out then o
                             else o'++"_"++n,
                             if i'==inp then i
                             else i'++"_"++n)
          in (Var_p new_o,s,Var_e new_i)) lls1
        lets'=map (\(s1,s2) -> (replace [inp] i s1,
                                     replace [out] o s2))
          lets1
        lets''=map (\(s1,s2) -> (replace [out] o s1,
                                     replace [inp] i s2))
          lets'
        lets'''=map (\(s1,s2) -> (s1++"_"++n,s2)) lets''
        (lls3,lets2,cnt2')=expand ls cnt2
    in
      ((lls2++lls3),(lets'''++lets2),cnt2')
  Switch _ -> error "Switch in PSF"
  Switch2 _ -> error "Switch in PSF"
  -- Eliminate arrow combinator >>>
  Seq (sf1,sf2) ->
    let v = "v_" ++ intToString cnt
        (lls,lets,cnt2) = expand ([ (Var_p v,sf1,Var_e i)] ++
                                   [ (Var_p o,sf2,Var_e v)] ++ ls) (cnt+1)
    in
      (lls,lets,cnt2)
  LocalTime -> error "expand : LocalTime in PSF"
  -- Don't touch other primitive signal functions.
  l' -> let (lls,lets,cnt2) = expand ls cnt
        in ((Var_p o,l',Var_e i):lls,lets,cnt2)

expand arg cnt = error "expand : unexpected argument"

```

```

-- Eliminates the primitive signal function localTime.

replace_t :: SigFn -> SigFn
replace_t sf = case sf of
  SigFn(n,i_,o_,sfs) ->
    let t_vars = find_t_vars sfs
        find_t_vars [] = []
        find_t_vars ((Var_p o,LocalTime,i):ls) =
          if o=="t" then (find_t_vars ls) else [o]++find_t_vars ls
        find_t_vars (l:ls) = find_t_vars ls
        replace_t_vars t_vars [] = []
        replace_t_vars t_vars ((o,LocalTime,e):ls) =
          replace_t_vars t_vars ls
        replace_t_vars t_vars ((o,sf,e):ls) =
          [(o,sf,replace_E t_vars "t" e)]
          ++ replace_t_vars t_vars ls
    in
      SigFn(n,i_,o_,replace_t_vars t_vars sfs)
  _ -> error "replace_t : wrong format signal function"

-- Auxiliary function to rename variables in an expression.

replace_E :: [String] -> String -> Expr -> Expr
replace_E vars n_str e = case e of
  Var_e str -> if elem str vars then
    let (fst,id,last)= findVar n_str
    in if last==" " then Var_e n_str
    else Expr n_str
  else e

  Expr str -> Expr (replace vars n_str str)
  Expr_e ev -> Expr_e (replace_E vars n_str ev)
  Expr_f (n,exp) -> Expr_f (n, replace_E vars n_str exp)
  Expr_p (e1, e2) -> Expr_p (replace_E vars n_str e1,
                             replace_E vars n_str e2)

-- Auxiliary function to rename variables in an event.

replace_Ev :: [String] -> String -> Ev -> Ev
replace_Ev vars n_str ev = case ev of
  MergeE (ev1,ev2) -> MergeE (replace_E vars n_str ev1,
                              replace_E vars n_str ev2)
  TagE (ev,e) -> TagE (replace_E vars n_str ev,

```

```

        replace_E vars n_str e)

-- Auxiliary function to rename variables in a string.

replace :: [String] -> String -> String -> String
replace vars n_str str =
    let (str1,v,str2) = findVar str
    in if v==" " then str
        else if elem v vars then str1++n_str++(replace vars n_str str2)
            else str1++v++(replace vars n_str str2)

-- Finds the variables in a string.

findVar :: String -> (String,String,String)
findVar str = findVar' False str " " " "

findVar' :: Bool -> String -> String -> String ->
            (String,String,String)
findVar' b [] id str1 = (str1,id,"")
findVar' b (c:cs) id str1 =
    if b then if (isAlpha c || isDigit c || elem c "_'") then
        findVar' b cs (id++[c]) str1
    else
        (str1,id,(c:cs))
    else if isAlpha c then
        findVar' True cs [c] str1
    else findVar' False cs id (str1++[c])

-- Filter variables in the input signal pattern.

expand_inp :: Patr -> [Name]
expand_inp (Var_p n) = [n]
expand_inp (Pair(p1,p2)) = (expand_inp p1) ++ expand_inp p2

-- Evolution functions corresponding to the variables in the output
-- signal (i.e global variables shared by locations). The event
-- variable is denoted as out_e if it exists.

expand_out :: Expr -> Bool -> Int -> ([EvFn],Maybe EvFn,Int)
expand_out exp top cnt=
    case exp of
        Var_e n -> ([EvFn_C (n, Expr "")],Nothing, cnt)
        Var_f _ -> error "functions are not allowed as output"
        Expr_e _ -> error "events are not allowed as output"

```

```

Expr_f (_,_) -> ([EvFn_C ("out"++intToString cnt, exp)],Nothing,cnt+1)
Expr_str -> ([EvFn_C ("out"++intToString cnt, exp)],Nothing,cnt+1)
Expr_p (exp',exp'') -> if top
  then let (x,_,cnt2) = expand_out exp' False cnt
        in case exp'' of
            Var_e evar -> (x, Just (EvFn_C (evar, Expr "")), cnt2)
            Expr_e ev  -> (x, Just (EvFn_C ("out_e", Expr_e ev)), cnt2)
            _          -> expand_out exp False cnt
  else let (x,_,cnt2) = expand_out exp' False cnt
        (x',_,cnt3) = expand_out exp'' False cnt2
        in (x++x',Nothing,cnt3)

-- Auxiliary function to parse patterns.

parseP :: Patr -> String -> [LetE]
parseP (Pair(p1,p2)) v =
  (case p1 of
    Var_p v1 -> [(v1,"fst "++v)]
    Pair (p3,p4) -> parseP p3 ("fst "++v) ++ (parseP p4 ("fst "++v))
  ) ++
  (case p2 of
    Var_p v2 -> [(v2,"snd "++v)]
    Pair (p5,p6) -> parseP p5 ("snd "++v) ++ (parseP p6 ("snd "++v))
  )

-- Condense a list of let expressions.

expand_lets :: [LetE] -> [LetE]
expand_lets [] = []
expand_lets ((v,exp):rest) =
  let rest' = map (\(v',exp') -> (v',replace [v] exp exp')) rest
  in (v,exp):(expand_lets rest')

intToString :: Int -> String
intToString i = if i<10 then [intToDigit i]
               else intToString (i `div` 10) ++
                  [intToDigit (i `mod` 10)]

-- Prints a hybrid system as a dot graph definition.

print_dot sf = do putStrLn "diGraph G {"
                  putStrLn "node [shape=box]"
                  putStrLn "rankdir=LR"

```

```

        putStrLn "rotate=90"
        putStrLn "size=\"10,7.5\" "
        putStr (print_hs (transform sf "sf"))
        putStrLn "\n}"

print_hs :: HS -> String
print_hs (n,locs,evs,(init_l,init_v)) =
    (foldl1 (++) "" (map (print_loc False) locs))++
    (foldl1 (++) "" (map print_arc evs))

-- Prints a single location as a node of a dot graph.

print_loc :: Bool -> Loc -> String
print_loc screen (LocS (n,vars, fs)) =
    let eol = if screen then "\n" else "\\l"
        inp' = inp vars
        out' = out vars
        ev' = ev vars
        lcl = local vars
        pr [] = ("","","")
        pr (f:fs) =
            let (str,n) = case f of
                EvFn_C (n,Expr "") ->
                    error ("print_loc : "++n)
                EvFn_C (n,exp) ->
                    (n++"(t)="++(pr_exp exp)++eol,n)
                EvFn_D (n,e1,e2) ->
                    (n++"(0)="++(pr_exp e1)++eol++
                     n++"(t)="++ pr_exp e2++eol,n)
                EvFn_E {var_e=n, when=e1, val=e2} ->
                    (n++"_pred(t)="++(pr_exp e1)++eol++
                     n++"_val(t)="++(pr_exp e2)++eol,n)
            (o,l,estr) = pr fs
        in if n 'elem' inp' then (o,l,estr)
           else if (ev' /= Nothing)
               then let Just evn = ev'
                    in if n==evn then (o,l,str++estr)
                       else if (n 'elem' out') then (str++o,l,estr)
                                                         else (o,str++l,estr)
               else if (n 'elem' out')
                   then (str++o,l,estr)
                   else if n 'elem' lcl then (o,str++l,estr)
                       else error "print_loc"
        (o,l,estr) = pr fs

```

```

nN = map toUpper n
on = n ++ " [labelloc=l, label=\" "++nN++"\\n"
inpstr = let on = "I n p u t :"+eol
        in case inp' of
            [] -> error "no input variable"
            [i] -> on++i++"(t)"++eol
            is -> on++
                (foldr1 (\s1 s2->s1++"(t), "++s2++"(t)") is)++
                eol
locstr = let on = "L o c a l :"+eol
        in case l of
            [] -> ""
            _ -> on++l++eol
outstr = let on = "G l o b a l :"+eol
        in case o of
            [] -> error ("no output variable:"+n)
            _ -> on++o++eol
evstr = let on = "E v e n t :"+eol
        in case estr of
            [] -> ""
            _ -> on++estr++eol
str = inpstr++locstr++outstr++evstr
in
    if screen then str
    else on++str++"\\n"

print_loc _ _ = error "print_loc only handles single locations"

-- Prints transitions of a hybrid system in dot format.

print_arc (l1,Pred str,l2,init) =
    let LocS (n1,vars1,_) = l1
        LocS (n2,vars2,_) = l2
        pr_init [] = ""
        pr_init (i:is) = case i of
            EvFn_C (n,Expr str) -> n++":="+str++"\\n"
                                ++pr_init is
            _ -> error "wrong format initialization"
        on = n1++" -> "++n2++" [label=\""
        arka = "\\n]"
        lab = str++"\\n"++pr_init init
    in
        on++lab++arka

```

Intended to be blank.

Bibliography

- [1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
- [2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [3] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.
- [4] R. Alur and T. A. Henzinger. Modularity for timed and hybrid systems. In *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *LNCS*, pages 74–88. Springer Verlag, 1997.
- [5] R. Alur and T. A. Henzinger. Modularity for timed and hybrid systems. In *Proceedings of the Eighth International Conference on Concurrency Theory (CONCUR)*, volume 1243 of *LNCS*, pages 74–88. Springer Verlag, 1997.
- [6] R. C. Arkin. *Behavior-based Robotics*. MIT Press, 1998.
- [7] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, April 1986.
- [8] R. A. Brooks. Integrated systems based on behaviors. *SIGART Bulletin*, 2(4):46–50, 1991.
- [9] A. Courtney. Frappé: Functional reactive programming in Java. In *Third International Symposium on Practical Aspects of Declarative Languages (PADL)*, March 2001.
- [10] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *2001 Haskell Workshop*, September 2001.
- [11] X. Dai and G. Hager. Specifying behavior in C++. In *International Conference on Robotics and Automation*, pages 153–160, 2002.
- [12] A. Deshpande, A. Gll, and L. Semenzato. The shift programming language for dynamic networks of hybrid automata.

- [13] H. M. 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.
- [14] C. 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, Oct. 1997.
- [15] R. J. Firby. Task networks for controlling continuous processes. In *Proc. of the Second Int. Conf. on AI Planning Systems*, Chicago, IL, 1994.
- [16] R. J. Firby. Modularity issues in reactive planning. In *Proc. of the Third Int. Conf. on AI Planning Systems*, 1996.
- [17] E. Gansner, E. Koutsofios, S. North, and B. Isci. Drawing graphs with dot, Feb. 2002. dot Language User’s Manual. AT&T Labs.
- [18] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. *Lecture Notes in Computer Science*, 1486:75–??, 1998.
- [19] P. L. Guernic, T. Gauthier, M. L. Borgne, and C. L. Maire. Programming real time applications with signal. In *Proceedings of the IEEE*, volume 79, 1991.
- [20] 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.
- [21] 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.
- [22] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
- [23] 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*, Minneapolis, USA, 1996.
- [24] F. Ingrand, M. Georgeff, and A. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
- [25] K. Konolige. Colbert : A language for reactive control in sapphira. Technical report, SRI International, June 1997.
- [26] K. Konolige, K. Myers, and E. Ruspini. The sapphira architecture: A design for autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9, 1997.
- [27] S. Lang. The xvision2 software system, May 2001. Master Thesis, Dept. of Computer Science, The Johns Hopkins University.

- [28] V. Lumelsky and A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Trans. on Robotics and Automation*, 31:1058–63, 1986.
- [29] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O automata revisited. *Lecture Notes in Computer Science*, 2034:403–??, 2001.
- [30] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600, pages 447–484, Mook, The Netherlands, 3–7 June 1991. Springer-Verlag.
- [31] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pages 149–178, 1992.
- [32] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30(2):181–202, 1993.
- [33] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell’02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, Oct. 2002. ACM Press.
- [34] H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling. In *Proceedings of PADL’03: 5th International Workshop on Practical Aspects of Declarative Languages*, pages 376–390. Springer Verlag LNCS 2562, Jan. 2003.
- [35] J. Ostroff. *Temporal Logic for Real-Time Systems*. John Wiley & Sons, 1989.
- [36] R. Paterson. A new notation for arrows. In *ICFP’01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.
- [37] I. Pembeci, G. Hager, H. Nilsson, J. Peterson, and D. Burschka. Functional Reactive Robotics: An exercise in principled integration of domain-specific languages. In *Proceedings of the 2002 ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Pittsburgh, PA, USA, October 2002.
- [38] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conf. on Robotics and Automation*, May 1999.
- [39] 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.
- [40] J. Peterson, V. Trifonov, and A. Serjantov. Parallel Functional Reactive Programming. In *Proc. 2nd International Workshop on Practical Aspects of Declarative Languages*, 2000.

- [41] S. Peyton Jones (ed.). Haskell 98: A non-strict, purely functional language. Technical Report RR-1106, Yale University, February 1999.
- [42] 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.
- [43] V. M. Santos, J. P. Castro, and M. I. Ribeiro. A nested-loop architecture for mobile robot navigation. *The International Journal of Robotics Research*, 19(12), December 2000.
- [44] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*, 1998.
- [45] P. Tabuada, G. Pappas, and P. Lima. Compositional abstractions of hybrid control systems. In *Proceedings of the 40th IEEE Conference on Decision and Control, Orlando, Florida.*, 2001.
- [46] Z. Wan. *Functional Reactive Programming for Real-Time Reactive Systems*. Ph.d. dissertation, Computer Science Department, Yale University, Oct. 2002.
- [47] 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.
- [48] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP'01)*, 2001.
- [49] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages (PADL'02)*, Jan. 2002.

Vita

Izzet Pembeci -- pembeci@cs.jhu.edu

Work Address: 338 Krieger Hall
The Johns Hopkins University
Baltimore, MD 21218
Tel:(410) 516-7036

Home Address: 2120 St. Paul Apt:3
Baltimore, MD 21218
Tel:(410) 347-1266

Date of Birth : 04/14/1973
Place of Birth : Ankara, Turkey

EDUCATION

Jan 98 - Aug 03	The Johns Hopkins University, Baltimore MD Ph.D. Computer Science
Sept 95 - Dec 97	Middle East Technical University, Ankara TURKEY M.S. Computer Science
Sept 91 - June 95	Middle East Technical University, Ankara TURKEY B.S. Computer Science B.S. Mathematics (Double Major Program)

RESEARCH INTERESTS

- . Functional Programming
- . Robot Languages
- . Computational Modeling of Turkish Morphology
- . Information Retrieval Techniques for Turkish
- . Complexity Theory, Self Organization