Reactive Queries in Mobile Ad Hoc Networks

Andoni Lombide Carreton*
alombide@vub.ac.be

Tom Van Cutsem†
tvcutsem@vub.ac.be

Wolfgang De Meuter
wdmeuter@vub.ac.be

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel, Belgium

ABSTRACT

Pervasive computing in mobile ad hoc networks requires that applications query their network environment for services and react to a plethora of events fired by other devices in that network responding to such queries. Current context-aware and event-driven architectures require the programmer to react to these events via a carefully crafted network of observers and event handlers, while inherently introducing complex concurrency issues. This paper proposes the integration of two techniques to solve these problems: ambient references and reactive programming. Ambient references are object-oriented communication abstractions that represent nearby remote objects in the mobile network and that make it possible to generate the events mentioned above. The reactive programming paradigm provides an abstraction over events such that the application can be written in a conventional programming style with explicit control flow.

Categories and Subject Descriptors

D1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages

Keywords

AmbientTalk, mobile ad hoc networks, event-driven programming, reactive programming

*Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

†Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

MPAC'08, December 1-5, 2008 Leuven, Belgium

1. INTRODUCTION

Because of the constant evolution of computing hardware, mobile computing devices with networking capabilities are becoming increasingly cheap, small and energy efficient. To reap the benefits of the resulting mobile networks, applications must be able to query their network environment for services and respond to changes in that environment in a timely fashion. State of the art context-aware applications are often conceived as event-driven architectures which consume events fired by a context-aware middleware framework. The events represent significant context changes which should percolate through the entire application, requiring a carefully crafted network of observers combined with complex synchronization code to deal with the inherent concurrency issues. By using these classic event-driven or publish/subscribe approaches, the application logic has to be scattered across event handlers which can be independently triggered at any point in time. The result is that the control flow of the application becomes very implicit in the source code.

When we look at the querying for services in mobile ad hoc networks, there are three types of network events that are of interest:

- The discovery of new services,
- the disappearance of services,
- the reception of replies resulting from a query over the network.

In this paper, we propose the integration of two mechanisms to allow applications that have to deal with these events to be written in a conventional programming style instead of an event-driven style. We will integrate them in the distributed, object-oriented language AmbientTalk, which is explained in section 2. The first mechanism consists of ambient references. Ambient references allow the broadcasting of messages to parts of the network represented as objects of a certain interface. Ambient references allow us to generate the three types of events mentioned above. They are discussed in section 3. In section 4, we show how we use the Reactive Programming facilities of AmbientTalk to hide the explicit events from the programmer and let the interpreter take care of the signalling of and reacting to these events. Reactive programming and the integration with ambient references is explained in section 5. In section 6 the limitations and assumptions of our implementation are discussed. Section 7 is our position statement and the last section concludes this paper.

2. AMBIENTTALK

AmbientTalk [11, 10] is a distributed programming language embedded in Java[1]. The language is designed as a distributed scripting language that can be used to compose Java components which are distributed across a mobile ad hoc network. The language is developed on top of the J2ME platform and runs on handheld devices such as smart phones and PDAs. Even though AmbientTalk is embedded in Java, it is a separate programming language. The embedding ensures that AmbientTalk applications can access Java objects running in the same JVM. These Java objects can also call back on AmbientTalk objects as if these were plain Java objects.

The most important difference between AmbientTalk and Java is the way in which they deal with concurrency and network programming. Java is multithreaded, and provides either a low-level socket API or a high-level RPC API (i.e. Java RMI) to enable distributed computing. In contrast, AmbientTalk is a fully event-driven programming language. It provides only event loop concurrency [6] and distributed objects communicate by means of asynchronous message passing. Event loops deal with concurrency similar to GUI frameworks (e.g. Java AWT or Swing): all concurrent activities are represented as events which are handled sequentially by an event loop thread.

AmbientTalk is designed particularly for ad hoc networks:

1. In an ad hoc network, objects must be able to discover one another without any infrastructure (such as a shared naming registry). Therefore, AmbientTalk has a service discovery engine that allows objects to discover one another in a peer-to-peer manner. Java interfaces act as the common pieces of information by means of which objects are discovered.

2. In an ad hoc network, objects may frequently disconnect and reconnect because of network partitions. Therefore, AmbientTalk provides fault-tolerant asynchronous message passing between objects: if a message is sent to a disconnected object, the message is buffered and resent later, when the object becomes reconnected. Other advantages of asynchronous message passing over standard RPC is that the asynchrony hides latency and that it keeps the application responsive (i.e. the event loop is not blocked during remote communication and is free to process other events).

The following code snippet illustrates how AmbientTalk can be used to discover a WeatherService component in the ad hoc network. Once discovered, the component is sent a message to retrieve the current weather.

```
when: WeatherService discovered: { |weatherSvc|
  when: weatherSvc<-getWeather("Leuven") becomes: { |info|
    // update weather information in the user interface
  }
}
```

The above code consists of two event handlers. The first event handler, declared by means of the **when:discovered:** control structure, is invoked when the language runtime discovers a WeatherService component. Here, WeatherService refers to a Java interface. The discovered object is accessible via the weatherSvc variable, which denotes a remote AmbientTalk object that wraps a Java component implementing the weather service. The syntax obj<-msg() denotes an asynchronous message send and represents a remote query.

When the query message is received by the remote weatherSvc object, that object's getWeather method is invoked. The return value of this method is used as the reply to the query. This reply is signalled asynchronously to the caller. The **when:becomes:** control structure is used to install an event handler that can process this reply. The return value is passed to this event handler (cf. the info variable in the example).

As can be seen from the above example, service discovery and replies of remote queries are represented in AmbientTalk as events that trigger the appropriate event handlers. While in this simple example the control flow remains apparent enough to understand, the control flow of large-scale event-driven applications can quickly become puzzling. In the following section, we show how the events of discovering new and detecting lost services can be made implicit, by means of what we call ambient references.

3. AMBIENT REFERENCES

When writing AmbientTalk code to query nearby services for data (e.g. all nearby temperature sensors in a wireless sensor network) using the language features discussed in the previous section, one often writes a recurring pattern of code to deal with the discovery and loss of nearby services while a query is executing, and to deal with gathering the replies from all respondent services. To ease the writing of multicast queries in AmbientTalk, we have introduced a novel data type in AmbientTalk, named ambient references [9].

Ambient references represent a collection of nearby services of the same type. This collection is constantly kept up-to-date with the proximate physical environment: newly discovered services are added to the collection, while unresponsive services are removed from it. This synchronisation with the environment must no longer be done manually by the programmer, but is instead done by the ambient reference itself.

Sending a message to an ambient reference causes this message to be multicast to all services in the collection. A message can also be annotated with an expiration period (in milliseconds). If a message has an expiration period, it will not only be multicast to all services in the ambient reference's collection at the time the message is sent, but also to any services discovered at a later point in time, until its expiration period has elapsed. Consider the following example query:

```
def sensors := ambient: TemperatureSensor;
whenAll: sensors<-getTemp()@Expires(5*1000) resolved: {
  |temperatures|
  // process the sensed temperature values
}
```

The keyword **ambient:** allows one to create an ambient reference given a Java interface. The variable sensors contains an ambient reference that refers to all nearby TemperatureSensor services. The message getTemp() is asynchronously multicast to these services with an expiration period of 5 seconds. This implies that the message may be received by all proximate sensors at the time it is sent, as well as to all additional sensors discovered within the next 5 seconds.

The **whenAll:becomes:** control structure allows the programmer to install an event handler that can be used to

[1]The language is available at `prog.vub.ac.be/amop`

gather the results of the query. Within this event handler, temperatures refers to an array containing the readouts of the sensors that replied. The event handler is triggered when the message's expiration period has elapsed. Ambient references support only weak delivery guarantees: some sensors may not have received the getTemp() message, and some replies to the message may have gotten lost or may arrive too late, in which case they are discarded.

The above example shows how ambient references relieve the programmer from having to deal explicitly with the events of discovery and loss of nearby services: ambient references transform these events into additions to or removals from their encapsulated collection. However, the programmer must still deal with the replies to the query in an event-driven manner by means of the **whenAll**:**becomes**: control structure. In the following section, we show how this event handler too can become implicit in the code, by means of reactive programming techniques.

4. REACTIVE PROGRAMMING IN AMBIENTTALK

The AmbientTalk interpreter was recently extended with reactive programming mechanisms to be able to write pervasive, context-aware applications more easily [7]. Reactive programming is a programming paradigm employed for various purposes such as animation [4], real time systems [12] and robotics [8], and can be introduced in existing languages such as Java [3]. A reactive system is built around the notion of time-varying values (called dataflows or *signals*). Changes to the values of these signals are automatically propagated to a network of dependent computations. Reactive programs construct this network either by explicitly wiring signals, or implicitly by calling functions or methods that take signals rather than ordinary values as arguments. Such functions or methods are named *lifted* functions or methods [2].

Whether one can apply a lifted function on a signal depends on whether the signal is continuous, i.e. whether it has a value at each point in time. Continuous signals (called *behaviors*) form the crux of the reactive programming model: by applying lifted functions on them, a dependent behavior is constructed transparently. Subsequent changes of the behaviors that were passed as arguments to the lifted function will be propagated transparently to the dependent behavior, allowing its value to be recomputed.

Signals which only carry events at discrete points in time are called *event sources*. Event sources provide a natural mechanism to interact with the real world, where events can be generated by input/output devices such as mice or keyboards.

Consider the following AmbientTalk code:

```
def minutes := seconds / 60;

system.println(minutes);
```

Assume that seconds is a behavior that is incremented by 1 every second. A new behavior minutes is implicitly created by passing seconds to the integer division function. Behind the scenes, a dependent computation is scheduled by the interpreter that will be re-executed every time seconds is updated. Finally, the minutes behavior is passed to the **system**.println function, adding another dependency to the dataflow graph. The result is that **system**.println will be recalled each time minutes signals an update, printing the new value on the screen every minute. Using this mechanism, one can construct an application that reacts to changes while preserving a conventional programming style instead of requiring the adoption of an event-driven style.

5. REACTIVE PROGRAMMING WITH AMBIENT REFERENCES

As discussed in section 3, using ambient references, event handlers must be used to collect the results of a broadcasted message due to the asynchronous execution of that message. The problem with using explicit event handlers is that the control flow of the application becomes much more implicit. In the literature, this phenomenon is also known as "inversion of control" [5]. Control flow is inverted because it is steered by an event loop, which lies outside of the control of the application itself. Also, because every event requires a separate event handler, control flow is often scattered across multiple event handlers. The order in which these handlers are triggered is not known. Finally, event-driven code is not modular: if a module's interface is event-driven, it often forces its client modules to have an event-driven interface as well. We will illustrate these difficulties by example in the following section.

5.1 Example: Ubiquitous Shopping Cart

Consider the following ubiquitous computing application. We assume a futuristic supermarket where all products on the shelves are RFID-tagged. The customers' shopping carts are equipped with an RFID-reader and a small computer screen. This screen is used to show, among others, the list of products in the customer's cart and e.g. the total price of all items in the cart (assuming that the RFID tag of the product contains a.o. its price). We assume that the screen updates this information once every three seconds.

In AmbientTalk, we can model the nearby RFID-tagged products as remote objects (we assume that products are simple local proxy objects whose attributes are constructed by physically reading the tag). As such, we can model all nearby products by means of an ambient reference. Part of the application running on the customer's shopping cart could then look as follows:

```
def priceModel := createPriceModel(); // create a new model
def pricelistView := createPriceListView();
def statisticsView := createStatisticsView();
// register views with model
priceModel.addView(pricelistView);
priceModel.addView(statisticsView);
```

We assume a typical model-view-controller architecture [1] in which the continuously updated prices of nearby products are represented as a model, and the two graphical user interface views are registered as views observing this model. Note that we want to maintain a separation of concerns between the views on the model and the model itself. In other words, the model should not know which views are registered on it. To this end, the PriceModel provides the following method to keep track of the views on itself in a views collection:

```
def addView(view) { views.add(view) };
```

By using a collection to store its views, the model does not have to know which views are actually registered on it. Within the PriceModel, the price of nearby products is queried periodically by means of an ambient reference:

```
// Assume that Product is a Java interface.
def products := ambient: Product;

def REFRESH_RATE := 3*1000; // in milliseconds

whenever: REFRESH_RATE elapsed: {
  whenAll: products<-getPrice()@Expires(REFRESH_RATE)
  resolved: { |prices|
    views.each: { |view| view.update(prices) };
  }
}
```

In AmbientTalk, the **whenever:elapsed:** control structure can be used to execute a block of code at regular time intervals. Here, we use this control structure to repeatedly (i.e. every 3 seconds) query the price of nearby products by sending the getPrice() message to the ambient reference. Note that the dependent UI views are manually updated whenever all replies to a query have been gathered (the expression views.each: is used to iterate over all registered views of the model). This is done inside of a **whenAll:resolved:** event handler because the getPrice() message is broadcasted asynchronously to all products in range and, as mentioned in section 2, does not return a result. Instead, to be able to process the results of the query, one should explicitly register an event handler that is triggered by the expiration of the query. In response to the update message, the price list view can update its displayed list of prices, and the statistics view can recompute e.g. the total price of products in the shopping cart.

The control flow of the example application is determined by three kinds of event handlers:

1. the **whenever:elapsed:** control structure repeatedly reacts to the fact that 3 seconds have elapsed and in response performs the remote query,
2. the **whenAll:resolved:** control structure reacts to the fact that the query's expiration period has ended, and in response updates the UI views,
3. the UI views react to this update (via their update methods) and in response redraw their UI.

As such, the application suffers from an inversion of control. For example, the updating of the views by the model happens only implicitly from the point of view of the first code snippet. Neither is it apparent from that code which view is updated first. In this particular application, the order in which views are updated is not so important, but in general this could become a problem when considering side-effects. Finally, the fact that the result of a query is updated asynchronously forces the application to use a Model-View-Controller pattern where the dependencies between model and views have to be manually encoded. In the code, the model is responsible for explicitly updating its dependent views. In the following section, we show how these dependencies can be managed automatically by the interpreter by making use of reactive programming techniques.

5.2 Reactive Queries

We now show how the result of queries can be made reactive by integrating ambient references with the reactive programming language facilities of AmbientTalk. Using reactive queries, the shopping cart application looks as follows:

```
def products := ambient: Product;
def pricelistView := createPriceListView();
def statisticsView := createStatisticsView();
def REFRESH_RATE := 3*1000; // in milliseconds

def prices := products<-getPrice()@Refresh(REFRESH_RATE);

pricelistView.update(prices);
statisticsView.update(prices);
```

Note that the broadcasted getPrice() message is now annotated with **@Refresh** rather than **@Expires**. The variable prices which contains the result of the getPrice query is now a behavior which initially denotes an empty array. The **@Refresh** annotation also causes the getPrice() message to be broadcasted repeatedly every 3 seconds to all nearby products. This causes the prices behavior to denote a new array of updated values every 3 seconds. Since prices is a behavior, it can be passed on to other functions or methods as a normal value. In the last two lines of code of the above code snippet, the views are explicitly updated with the prices behavior which creates a dataflow dependency between the behavior and the UI views. This means that whenever the prices array is recomputed, the update methods of the views are implicitly called as well.

One advantage of reactive queries is that the application is no longer forced to use an explicit Model-View-Controller pattern. The task of a Model to update its dependent Views is now replaced by the implicit dataflow dependencies on behaviors. As a result, the above code no longer needs to explicitly update the views every time a new query result is computed. This is because the dataflow dependencies are immediately created in the initialization code of the views by passing a behavior to the update methods of the views. This causes the interpreter to handle the updating process by reacting to the underlying events and invoking the dependent functions and methods, instead of requiring a Model abstraction for this purpose. Using reactive queries, there is no inversion of control: there is one dominant control flow which establishes dataflow dependencies by creating and passing behaviors. Also, note that the order in which the views are updated is immediately apparent from the code.

To conclude, integrating ambient references with reactive programming allows the results of queries over the network to be collected into a behavior that is automatically synchronized with the environment. Ambient references provide an abstraction over the events of appearance and disappearance of services in the network, while the reactive programming system provides an abstraction over the events generated by the reception of results of asynchronous queries.

5.3 Implementation

There is a working implementation available of the constructs proposed in this paper[2]. In this section, we discuss some of the features and technical aspects that may be relevant when using the system.

5.3.1 Handle interface

When broadcasting a message using an ambient reference, the programmer can specify to get a so-called *handle* object as the return value, rather than a behavior. A handle object offers a richer interface to manage the executing query and the event processing by the reactive programming system.

[2]The virtual machine with the required libraries can be downloaded here: `http://prog.vub.ac.be/~alombide/AT2-Reactive.zip`

It provides a cancel method to cancel the repeated broadcasting of the message. This can be used to control the broadcasting behavior of the ambient reference (e.g. stop broadcasting when the battery power of the host device is low). Furthermore, handles provide a snapshot method that returns a collection of results that is a snapshot of the query, i.e. a static collection that is not updated automatically. Finally, one can access a special type of value (called a *future*) on which event handlers can be manually registered to gather the return values of broadcasted messages.

5.3.2 Reactive Programming Interface

In AmbientTalk, a behavior can be created from an *event source*. Event sources represent streams of discrete events over time. An event source can be created as follows:

```
def [ eventSource, notifier ] := makeEventSource();
```

makeEventSource() returns an array of two objects: the eventSource object is the event source itself and the notifier object is a special object that can be used to push new events on the event source. This is done as follows:

```
notifier <-notify(newValue);
```

Finally, the **hold**: primitive can be used to convert an event source to a behavior. At any point in time, the value of the behavior is the last value on its associated event source stream.

5.3.3 Implementing Reactive Queries

We now describe how the behavior returned by an ambient reference is implemented in terms of the reactive programming interface discussed in the previous section. The code excerpt below is part of the implementation of reactive ambient references.

```
def [theResultsEventSource, theNotifier] := makeEventSource();
def theBehavior := hold: theResultsEventSource;

def createResultsBehavior() {
  def registerNextObserver() {
    whenAll: handle.future resolved: { |returnValues|
      theNotifier<-notify(returnValues);
      // register observer for the next iteration of the query
      registerNextObserver();
    };
  };
  registerNextObserver();
  theBehavior;
};
```

In the underlying implementation, a **whenAll**:**resolved**: event handler (as used in the first example in section 5) is repeatedly installed on the future of the broadcasted message. In this event handler, the notify method of the notifier object associated with the event source is called with the received results of the broadcasted message. This causes the behavior (and any depedent computations) to be updated.

6. LIMITATIONS

In this section, we discuss the limitations of our implementation and the assumptions on which it relies. First of all, the system was not designed to respond in real-time to events. This becomes apparent when a large expiration period for a query is used: there is no possibility to react to individual events before the expiration period has elapsed. It is the programmer who has to strike the balance between giving the query enough time to gather the results and refreshing the results quickly enough to keep the application responsive. Related to this issue is the fact that on devices powered by batteries, the continuous broadcasting of messages may be too power hungry in situations where frequent querying of the environment is necessary.

Furthermore, a few assumptions have been made with regard to deployment. Services for instance, must be explicitly exported as AmbientTalk objects. For this, we assume an AmbientTalk virtual machine (on top of a Java virtual machine) on each device. Of course, this is a relatively heavyweight setup. In applications making use of RFID technology, such as the example of the ubiquitous shopping cart discussed in section 5.1, one cannot assume that there is an AmbientTalk virtual machine running on each RFID tag which is able to respond to messages received by other devices. For applications like this, we use an intermediary device (a regular PC equipped with an RFID reader in our case) that scans the RFID tags and returns proxy objects that represent the scanned tags in response to queries.

Finally, the naming and discovery of services happens via Java interfaces. We make the underlying assumption that the name of such Java interfaces represents a unique service and is known by all participating services. This discovery mechanism also does not take versioning into account explicitly. For example, if the WeatherService from the example in section 2 is updated, older clients may discover the updated service, and clients that want to use only the updated service may still discover older versions. Clients and services are thus themselves responsible to check versioning constraints.

7. POSITION STATEMENT

Pervasive applications in mobile ad hoc networks require the frequent querying of the network environment for nearby services. The result is that nearby devices generate events, more specifically the appearance and disappearance of services and their responses to the queries, that are delivered to the application at any point in time. These queries must somehow be launched by the application and the asynchronous replies have to be somehow gathered and be acted upon. This calls for an event-driven architecture. State of the art event-driven middleware allows the application programmer to specify how to react to these events, but in doing so requires that the whole application is structured around event handlers causing an inversion of the control flow.

Our position is that we want to keep the event-driven nature of the middleware and the applications built on top, but to change the interface between both by replacing an interface based on event handlers with one based on reactive programming (behaviors). As a result, we can keep the responsiveness of a traditional event-driven system while avoiding the detrimental effects of inversion of control. In other words, reactive programming acts as a layer of abstraction over the bare events generated by the middleware allowing the control flow of applications to remain explicit.

8. CONCLUSION

We have shown that ambient references, which are existing abstractions in the AmbientTalk language, provide an object-oriented representation of parts of the network and provide a means to launch queries over these parts of the

network by broadcasting asynchronous messages to groups of remote objects. Ambient references handle the appearance and disappearance of remote services in the network transparently, but still require the programmer to manually implement event handlers for processing the asynchronous replies.

Reactive programming is a known programming paradigm supported by the AmbientTalk language that allows to express programs that have to deal with external events using a conventional programming style (i.e. with an explicit control flow managed by the interpreter). In this paper we have shown how ambient references and reactive programming can be unified, such that applications having to query the network using asynchronous communication can be implemented in a reactive programming style with explicit control flow, as opposed to an event-driven style where control is inverted. The advantage of reactive queries is that they keep the responsiveness of a purely event-driven approach while preventing an inversion of control of their client code.

9. REFERENCES

[1] S. Burbeck. Application programming in smalltalk-80: How to use model-view-controller (MVC). University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive. Available at: http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[2] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In P. Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, 2006.

[3] A. Courtney. Frappé: Functional reactive programming in Java. In *Third International Symposium on Pratical Aspects of Declarative Languages (PADL)*, March 2001.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, 1997.

[5] P. Haller and M. Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer, 2006.

[6] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In R. D. Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[7] S. Mostinckx, A. Lombide Carreton, and W. De Meuter. Reactive context-aware programming. In *Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008)*, volume 10 of *Electronic Communications of the EASST*. DisCoTec, June 2008.

[8] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *First International Workshop on Practical Aspects of' Declarative Languages (PADL)*, January 1999.

[9] T. Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[10] T. Van Cutsem, S. Mostinckx, and W. De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages Systems & Structures*, 2008. To appear.

[11] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*, pages 3–12. IEEE Computer Society, 2007.

[12] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP'01)*, 2001.