Contextual Interfacing: A sensor and actuator framework

Kasper Hallenborg

Maersk Mc-Kinney Moller Institute
University of Southern Denmark
Odense M, 5230, Denmark
hallenborg@mip.sdu.dk

Abstract. The purpose of this article is to describe a middleware framework for interfacing sensors and actuators, which provides abstract and useful contextual information to applications. The framework is divided into three layers with different abstraction levels, and an integration layer. The framework is evaluated, exemplified, and further described by practical implementations using the framework.

Keywords: Sensors, context, abstraction, framework, pervasive computing

1 Introduction

Human perception is the mental processes, which deals with gathering and interpreting context information. In ubiquitous and pervasive computing, we face a similar task, when we try to establish a realistic relation between context and behavior of the application. Sensors are the analogy to human senses and the key sources for providing information from the environment. Interfacing of sensors and actuators is a common problem in application development. Each sensor and actuator has their own protocol and connection type, which makes it hard to implement a transparent interface from an application point of view.

For human-interactive, context-aware applications, the response time for a sensor or actuator has lower priority, thus the number of abstraction layers could be increased for the interface. With an increased number of layers in the interface, it is easier to reuse and generalize specific parts of the interface scheme.

2 Related work

Dealing with sensors is not a new trend in software applications, but the number of sensors providing information to the applications has increased rapidly over the last years, and they are a must for creating the killer applications of pervasive computing, which truly will revolutionize our impressions of computers, if they follow the spirit of Mark Weiser's *Calm Technology* [19]. Thus, abstracting and interpreting sensor information is and will be a crucial task of pervasive applications, which means that standardizing sensor integration and context interpretation will be an important part of most applications.

Anind Dey's *Context Toolkit* [1] is among the most cited works in this area. The Context Toolkit hides the details of the sensors behind *context widgets*, which encapsulates the code for communicating with the hardware sensors, and provides a standardized API for retrieving the data.

Another approach is taken by Schmidt et al [16, 17], who in a layered architecture hides the context retrieval process. It consists of three elements; *Sensor*s, which output is regarded as a function of time, *Cue*s, which combines and abstract the current and previous data from a particular sensor, the *Context* layer combines the input from all Cues to a current situation of the context, and finally the *Application* layer holds mechanisms to perform actions, when entering, leaving, and operating in a given context state.

Other approaches with focus on contextual integration and abstration are the CALAIS architecture [10], the Ektara [14] by MIT Media Lab, the Context Information Service (CIS) proposed by Pascoe [12]. More recently, Hackmann et al [13] have proposed a simple pattern-like architectural middleware (CONSUL) for providing context-awareness to application.

3 Approach and Overview

Basically, the framework is best described as a middleware software component with the purpose of adapting between hardware sensors and actuators, and context-aware applications. As a consequence of the finite computing capabilities, interfaces to the analog world can to some extent always be regarded as discrete. Typically we sample the environmental variable at a random point in time and regard the output as valid until the next sample. As the application domain of the framework is intended for context gathering in human-like perception processes, the intelligence built into the systems elsewhere should make the systems fault-tolerant to a few missing values from the environment. Thus *synchronization* must be the first aspect of dealing with sensors and actuators. In the layered model of the framework, figure 1, the synchronization layer is illustrated at the bottom, controlling the directed access with hardware interfaces of the sensors and actuators.

Requesting a few sensors values is far from the fascinating illusion of tomorrow's pervasive systems, and responding to changes of a single environmental variable is also on the edge of the definition of being context-aware. Similar to human perception, we would require the ability to combine various stimuli from different sources to a whole representation of a single impression. The next level of the framework is named *aggregation*, and covers not only combination of data from various sensors, but also aggregating multiple data from the same sensor to a single value, such as averaging over the last 2-3 values from a temperature sensor. Combined sensors do not necessarily have to provide mutual exclusive information, for instance a device could be equipped with two or more positioning sensors, which could be combined and prioritised to a single position, based on the situation of the device, such as indoor and outdoor.

Raw sensor data give little or no meaning to most users, and some information from the sensor might be irrelevant, such handshaking and status signals. E.g. GPS signals contains a lot of information, which is irrelevant, if only a "best guess" of the current position is requested. Beside from a second option to filter data, the *abstraction* level of the framework allow the user to convert the data to more meaningful information.

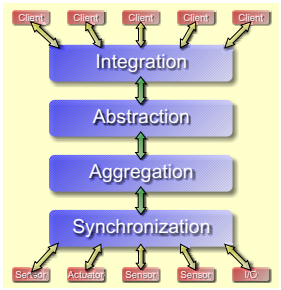


Fig. 1. Layered model of the sensor framework, composed by the four required components; Synchronization, Aggregation, Abstraction, and Integration

Last but not least, integrating context information into applications is perhaps the biggest challenge towards a generic approach for using context. Several previous attempt for unifying context integration exists, such as a generic context server [15]. We will provide a number of standardized interfaces for integration with existing technologies. Summarizing,

Synchronization grasp the current data from deployed sensors, or forwards command to actuators.
- General interfaces for standard purpose I/O interfaces.
- Support interest from multiple clients.

Aggregation provides means for combining primitive and related sensor data to a joint structure expressing a more cohesive stimuli.
- Unlimited and unconstrained means for combing all kinds of sensors.
- Parts of the structure could be reused for different purposes.
- Option to prioritize between overlapping sensors or sensors with common characteristics.

Abstraction interpreting the raw sensor data to more meaningful information for both humans and applications.
- Provide a common data structure for easing the integration of sensor and actuator information with applications.
- Provide a set of common structures for general context information, eg. location data.

Integration provides means for integrating the sensor framework with existing software technologies.
- Support a number of general technologies for integration with applications and services.

4 Synchronization

As stated above the purpose of the synchronization layer is to retrieve or send data from or to the hardware or logical devices, respectively. Retrieving data from sensors could either be initiated by frequently polling by the client or the interrupts from the hardware sensors sending event notifications through the interface. As one of the design criteria is to support a broad range of client functionality both mechanisms must be supported. Thus, besides the general methods for requesting data from a higher level in the framework hierarchy, the synchronization level should also be able to generate events and forward them upwards. Monitoring the outdoor temperature is a typical example of polling the sensors now and then, whereas triggering contacts in an alarm system justifies the need for event generations as well. The framework must support both methods, even it is commonly agreed that the notification model is preferable with respect to performance, but for some applications the setup of listeners, event generators, and event handlers requires an inappropriate amount of resources.

The other important functionality of the synchronizer is to support multiple clients' interest in the same data. The usual approach would be to discard the data after they have been retrieved by a client, and let the client to decide, which data needs to be stored. Supporting multiple clients raises the dilemma of making the same data available to all clients or regards each invocation from clients as a single request. As the first approach is the most general, we will go for that solution, as the second approach easily could be implemented on top of that by discarding all current data and request a new value.

4.1 Multiple client support

The second dimension of customizing the synchronizers is the support for multiple clients. It is not mandatory for a synchronizer to support multiple clients, based on the arguments of constrained devices and simple applications with no required support. Transparency should also be maintained, so a simple synchronizer could be substituted with a multi-client synchronizer, without affecting the existing code. It has one strong implication for the interface of the synchronizer, as most object-oriented programming languages do not provide access to the caller from the callee, without a passed reference to the caller. Identification of the caller is required in order to select and return the right data, and to clean up and maintain the repository of stored data.

4.2 IO and logical devices

Communicating with the real devices is one of the boundaries, where the framework has to fit with existing technologies and defined standards we typically cannot change or influence, both caused by physical limitations and the extensive distribution of sensors. The interface to the devices should be kept simple for the sake of generality; basically we are only interested in sending and receiving a chunk of data to and from the device. Configuration of ports and other details of

the communication protocol, settings, etc. should be hidden in the implementations of the interface, and would not be under control from the synchronizers, as transparency should secure that synchronizers would not care about the device is connected via a serial port, USB, WLAN, or a file-object for demo purposes.

5 Aggregation

Grouping information is, if not explicitly stated, a commonly accepted way to cope and deal with excessive amounts of information, if not the only one. More or less consciously we do it all the time, and it helps us to hide the unimportant details for a particular matter. Instead of describing everything by their atomic components, a cumulative identifier simplifies the model. In a normal conversation almost anybody would know, or a least have an idea of, what you mean by a house. So instead referring to four walls of bricks and windows, and a roof, where bricks are small boxes of burnt clay, windows are rectangular boxes of glass with a frame, which is ... , etc. Instead we have this composite identifier of a house, which makes conversation possible, but we have to accept that the other person do not necessarily picturize the same house, if not described further.

From this example it is also clear that various components, which are part of the complete composite structure, may themselves be identified entities of importance to others or in another situation, such as the window being a window without carrying about being part of a house. In this way we have different component or details of the composite, which has certain interest for different people or purposes, which again could include parts that are important to others.

So the purpose of the aggregation layer is to collect and present data from synchronizers in a meaningful way for the abstraction layer. The intension is to relate all data, which are required to construct a client-ready data structure in the abstraction layer, by only referencing a single aggregator.

Data from the aggregator may still be represented in the form they are extracted from the synchronizers, but a conversion could optionally be applied, such as error correction and filtering, corresponding to the nature of human perception, where only a few out of millions stimuli reach the memory matching processes that defines our behaviour.

Finding a data structure to hold the retrieved information in not an easy task, as both type or name of the data, and the data itself should be accessible. The approach taken is to compose a `Map` structure, where combined information from one or more sensors could be separated to individual entities in the mapping.

An aggregator is intended to communicate with only one synchronizer, making event propagation much easier, so aggregators must be able to combine their data to a single entity, which can be addressed from the abstraction layer. The *Composite* pattern [4] is a simple mean to combine aggregators in an unlimited tree-structure and keep a simple interface to the abstractors, which cannot tell if a complex structure or just a single aggregator is responding.

A `CompositeAggregator` provides a simple approach for combining data from a list of children into a single mapping. Composite aggregators could com-

bine different positioning measurement into single location information. Consider the case, where your device has two means to determine the location, such as a GSM phone, which could the determine the position based on the cell-ID or even a more advanced triangulation approaches. The phone could in addition be equipped with some sort of Bluetooth means for accurate locale indoor positioning.

The two methods of the `Aggregator` interface return and set the translator of the `Aggregator`, respectively. By this mean, it is extremely simple for the user to customize the `Aggregator` for a specific purpose, only an `AggregatorTranslator` has to be provided, converting data to and from the aggregation layer.

6 Abstraction

10.3 kΩ of a 20 kΩ LDR or NTC resistor may give sense to some, but most people would have no clue, and even technical skilled would have no idea of how it relates to lightning and temperature conditions, respectively. It depends very much on the scale and linearity of the sensors, just to convert the data into more familiar scales, such as *Lux* and *Celcius*. Or consider the following output from a sensor

```
$GPRMC,095211.808,A,5524.2563,N,01024.0460,E,2.03,324.78,230702,,*03
$GPGGA,095212.808,5524.2568,N,01024.0448,E,1,04,4.4,45.8,M,,,0000*3F
$GPGSA,A,3,20,11,01,07,,,,,,,,,7.9,4.4,6.6*38
```

you cannot gain much information from the listing above, if you are not familiar with the NMEA 0183 standard used by the GPS sensor.
There is no need for software designers to deal with these kind of low-level details, every time they request interaction with the environment, and therefore have to interface sensors. By abstracting the sensor specific information, we can bring sensors and actuators into play, with only little knowledge of the underlying hardware.

The purpose of the layer is to compose appropriate and client-ready data based on the data retrieved from the aggregation layer, and send it to clients through the integration layer. No further conversion or adaptation of the data should be performed in the integration layer, just integration and interfacing with various devices or protocols should be handled. Data can either be retrieved or put through the abstraction layer by the methods `getData` and `putData`, respectively.

Translators are used to convert data to and from the abstraction layer, in the same manner as the aggregation layer. The `AbstractorTranslator` is, similar to the `AggregatorTranslator`, defined as a static inner interface of the `Abstractor`. As for the aggregation layer, the purpose of using translators is to simplify the framework for the user, so just a simple translator implementing the conversion methods have to be provided in combination with the default `Abstractor` implementation.

Sticking to the focus of the framework as middleware for ubiquitous systems, one sensor (hardware or logical) would rarely be enough as the systems

evolve. The intension is to let the framework compose the full context conditions the application should respond to. Instead of the application composing its own weather condition from e.g. a temperature and rain sensor. Separating the aggregation and abstraction allow us to have different abstractions of the same aggregated structure. Abstractors could reference any node in the tree formed by the aggregators, so parts of the tree could have it own interpretation for one or more clients.

7 Integration

You can gather and interpret all the information you want, but if you do not make them available to users and try to fit their requirements for collecting such information, you will never reach more than your own test applications. Thus at the integration level of the framework, we have reached the upper boundary of the middleware, where we have to fit to existing standards in order to associate with most users, and this is not an attempt to be just a good sales guy. We could easily wrap the data and drop them into our own context server, but first and foremost that have not been the focus of the work conducted on the middleware framework. On the other hand there is a number of existing means to integrate or provide context information to users, and as closer we approach a general standard des easier will the integration be.

The intension of the integration layer is not further to manipulate or abstract the data received from the sensors, just to make is easier to provide the data in various ways, such as services in networks or integrated directly into applications. Our initial and main focus for integration have been the Jini^{TM} Technology, therefore the standard implementation provides default implementation of integrators and adaptors for the Jini community, but working and experimenting with other networking architectures or similar service systems, will constantly extend the number of provided adaptors and supported integration methods. By this mean integration with the Elvin Messaging system [2] has been implemented as further described in the sections of the examples.

Integrating directly with applications would typically be in terms of binary Java objects or standardized data structures, such as XML documents. Following the intensions of the framework these data should be provided in abstract and general formats from the abstraction layer, and the integrator might only have to wrap these information and make them accessible for the applications, for instance in terms of Java Bean objects.

8 Examples

We will look at three examples for testing different aspects of the framework, which might look rather independent, but in the end they will be grouped together to finally test those aspects of the framework.

The first example is to integrate a GPS sensor and provide positioning information in a general format and make it accessible to interested clients. The

example is a simple test of the most basic functionality of the framework, which will illustrate the minimal work that have to be done by the user. The next example is controlling a RCX brick from a PDA. Abstract and human-friendly commands should be translated and send to the RCX brick, for control of attached sensors and motors. The last example is the integration of the Smart-Its, a general sensor-board with multiple sensors, that will focus on testing the aggregation layer of the framework. Finally, all three examples will be combined to a single robot with a control part consisting of a PDA and a RCX brick, which will serve the purpose of testing reuse and combination of several independent setups in the framework.

9 Example: Interfacing GPS

Location is arguably the most important contextual information for most pervasive and ubiquitous applications, thus interfacing a positioning device is a relevant example for testing the framework.

A number of different positioning devices exist, but besides cellular-information of mobile phones, the GPS (Global Positioning System) is by far the most prevalent. Most common receivers use a simple serial protocol based on the NMEA sentences [11], so these sentences have to be parsed and combined in order to give some useful information for the application layer.
The basic purpose of the test is to retrieve and interpret data from a GPS receiver, which basically could be done with any GPS receiver that allow us to access the NMEA sentences. The challenges of deploying it to a PDA and let it control the receiver and framework components are minimal and not given any special focus during this test, so it could be mirrored to a laptop without any changes, except for a recompilation of the native part.

One way to provide location information in a general structure could be the SLO Protocol [7], which is one of the implemented standards in the framework to demonstrate the implementation of abstraction translators. It extends a standard XML translator that sets up all the basic for producing XML documents.

10 Example: Interfacing Lego Robots

In the recent years Lego has merged computer power and the standard bricks to enhance the intelligence of the toys. Special bricks can be programmed with some functionality. The focus of this example is to send simple immediate commands to the firmware in the RCX brick, such as start and stop of a motor, setting the speed of the motor, retrieving sensor inputs, checking the battery level, or playing a sound.

The simple approach is to control the RCX computer via the IR-tower, which is part of the Lego Mindstorm set. Our main objective is to control the RCX through an IR port of a PDA, therefore additional `IODevice` implementations have to be provided for the PDAs. We agreed on a setup, where a PDA with a built-in IR-port is mounted on the robot and the IR ports of the PDA and the

RCX are aligned. The PDA would be wireless connected to the main computer using wireless LAN. The protocol for communicating with the brick is somewhat challenging to the framework, because RCX opcodes and messages are wrapped into packets with headers and checksums. To complicate it even more, after each byte send (beside the header) the same byte must be resend negated.

Using the IR port of a PDA is a lot more complicated than using a serial connection, as we have to take control of the hardware pins on the controller boards using special registers of the PDA. We have successfully implemented communication with Cassiopeia E-200 and IPAQ H5550 PDAs.

10.1 Aggregation and abstraction

In the aggregation layer we handle the translation of RCX requests and commands into real RCX messages with header and checksum, and responses from the RCX are unwrapped, and the information is provided in a mapping structure to the abstraction layer. It is also the task of the aggregator to check messages and responses from the RCX, that they fulfil the required and expected format.

For the example we have implemented two abstractors, one which represent a Jini service object holding the full state of the RCX. The other abstractor generates Tickertape [3] messages for the Elvin messaging system, which are presented in a Tickertape application, where only the responses are translated into messages and commands.

11 Example: Interfacing Smart-Its

The Smart-Its from Lancaster University consist of a core board, which contain power sources, a programmable microcontroller, memory, and a wireless communication chip. The core board has a socket for add-on boards, which in our case is used for the general sensor add-on board containing 5 different sensors; Light sensor, motion detector, touch sensor, temperature sensor, and dual axis accelerometer.

Tailoring the layers Retrieving the information from the Smart-It base-station is extremely easy, as the standard code sets up the Smart-Its to put the sensor data over the serial port line by line. Thus, the code up to the aggregation level would be identical to that of the GPS receiver, except for the com port settings of the `IODevice`; 115,200 bps, 1 stopbit, 8 databits, and no parity.

Beside the important task of retrieving the data for the sensors through the synchronizers, the aggregator has to compute these changes based on the history of sensor data.

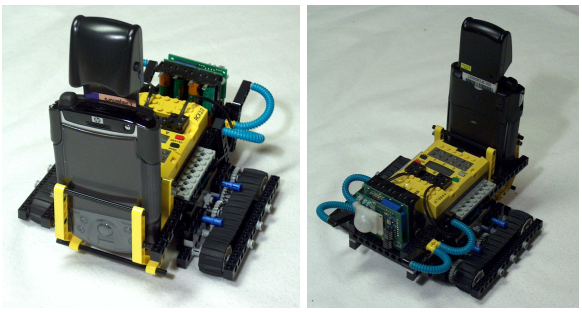
Aggregation layer The Smart-It sensors provide two types of values, either booleans from touch and motion sensors or integer values from the other sensors. The integer values can be considered as a discrete function of e.g. time

or number of measurements, thus we can apply numerical methods to compute properties for a similar or fitting continuous function, which is much easier to handle computationally and relate to, and predictions of future values could also be possible. In this example we will only focus on a number of simple properties for the function that can be useful to determine how extensive a change in the measured value is.

Abstraction layer Abstracting data from a general sensor board as the Smart-It to a united context descriptor is almost impossible for a general purpose, as the domain of context-aware application is so broad. We could tailor the abstraction of the Smart-It's data, to collect environmental context information for the robot only. Similar to the data abstracted from the GPS sensor to an XML document of the SLO protocol, we have defined an XML scheme with generalized Smart-Its data, holding the current values of the sensors and optionally the changes to the state of each sensor.

12 Example: Putting it all together

It may look like a weird artefact with superfluous technologies attached, but the challenge was to compose an artefact with various capabilities from existing technologies.



(a) Front (b) Rear

Fig. 2. The SmartRobot with the RCX computer, a IPAQ H5550, GPS receiver, and a Smart-It attached.

To give a brief overview of the application of the combined technologies in this example, a Lego robot will drive around and report various information about the environment, it is passing through, and it can be controlled by the user via simple commands. We will use the TickerTape messaging board of the Elvin message system to present these reports to the user, and allow the user to control the RCX robot via simple commands. The Smart-It sensor-node is also attached to the robot and a base-station is attached to a host computer, which runs the

example of the Smart-It and produces a XML document that is made available in a repository accessible by the IPAQ via WLAN. By this means the XML document becomes a source of input to the robot, which could be *sensed* by an XML parser.

12.1 Compose the primitive aggregators

We have chosen the easy approach of combining primitive aggregators that each represent a channel of context information from lower layers. The standard `CompositeAggregator` simply merge the mappings of child aggregators to a single mapping forwarded to the abstractors, ignoring overlaps in keys (last child overwrites first). Extending this algorithm prioritizing between identical or similar context information could be introduced, e.g. for the robot we could exclude the light sensors of the Smart-It, if light sensors were attached directly to the RCX or reverse.

12.2 Chatting with Elvin^{TM}

The objective of the SmartRobot is not to do any fancy tool work in the field, but just wandering about and report experiences from and about the environment it is passing through. The application can be seen as an extension of the RCX example, where commands are sent back and fourth between the RCX robot and the TickerTape message-board of the Elvin messaging system. Instead of simple and informative command-like messages, we have extended the abstractor to compose a number of more descriptive messages, that can report about changes in the environment, e.g. the temperature decreases/increases, and GPS information can, when available, be used to set a picture in memory of the current state, so messages could sound like:

> *... last time I was near this location, it was warmer and lighter, but my accelerometers tells me that I'm still not shaking ...*

13 Conclusion

We have designed and implemented a general framework for interacting with sensors and actuators. The default implementation and the core framework, provides a number of way to setup the communication with the hardware, and retrieval of data in particular. We support asynchronous multiple-client interests of the retrieved information. Customization of the hot spots of the framework in easily customizable through simple translators, which translate up and down between the data structures of the different levels in the framework. We have implemented a couple of examples to test and validate various aspects of the framework with great success and only minor changes to the core framework. We hope the supported technologies constantly will grow through different examples with the framework. We will continue working on the framework structure and enhance some of the provided tools for better and easier integration with various applications.

References

1. A. K. Dey. *Providing Architectual Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, November 2000.
2. DSTC. Elvin - content based messaging. http://elvin.dstc.edu.au/.
3. DSTC. Tickertape. http://elvin.dstc.edu.au/projects/tickertape/.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
5. H. Gellersen. The smart-its project. http://www.smart-its.org/, 2001.
6. J. Hightower, B. Brumitt, and G. Borriello. The location stack: A layered model for location in ubiquitous computing. In *In Proceedings of the 4th IEEE Workshop on mobile Computing Systems & Applications (WMCSA 2002)*, pages 22–28, Callicoon, NY, USA, June 2002. IEEE Computer Society Press.
7. M. Korkea-aho. *Location Information in the Internet*. PhD thesis, Helsinki University of Technology, October 2001.
8. Lego^{TM}. The offical lego mindstorms website. http://mindstorms.lego.com/.
9. L. Merk, M. S. Nicklous, and T. Stober. *Pervasive Computing Handbook*. Springer Verlag, January 2001.
10. G. J. Nelson. *Context-Aware and Location Systems*. PhD thesis, Clare College, University of Cambridge, United Kingdom, January 1998.
11. The national marine electronics association (nmea). http://www.nmea.org. NMEA 0183 standard.
12. J. Pascoe. *Context-Aware Software*. PhD thesis, Computing Laboratory, University of Kent at Canterbury, August 2001.
13. G. H. C. J. J. Payton and G.-C. Roman. Supporting generalized context interactions. In *Proceedings of the 4th International Workshop on Software Engineering and Middleware*, Linz, Austria, September 2004.
14. A. S. P. Richard W. DeVaul. The ektara architecture: The right framework for context-aware wearable and ubiquitous computing applications. Technical report, The Media Laboratory, MIT, February 2000.
15. D. Salber and G. D. Abowd. The design and use of a generic context server. Technical Report GIT-GVU-98-32, Georgia Institute of Technology, GVU Center, College of Computing, 801 Atlantic Drive, Atlanta, 1998.
16. A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde. Advanced interaction in context. In *Proceedings of the first International Symposium on Handheld and Ubiquitous Computing (HUC99)*, volume 1707, pages 89–101, Karlsruhe, Germany, 1999. Springer.
17. A. Schmidt and K. V. Laerhoven. How to build smart appliances. *Personal Communications, Special Issue on Pervasive Computing*, 8(4):66–71, August 2001.
18. A. M. R. Ward. *Sensor-driven Computing*. PhD thesis, University of Cambridge, August 1998.
19. M. Weiser and J. S. Brown. The coming age of calm technology. Technical report, Xerox PARC, October 1996.