The Agora software implementation
A system for development of smarter devices

Johannes Jansson

UmeåUniversity, Sweden
Queensland University of Technology, Australia

October 27, 2003

Umeå University, Sweden
Queensland University of Technology, Australia

Supervisor UMU:	Dr. Per Lindström
Department:	Computing Science
Supervisor QUT:	A/Prof. Joaquin Sitte
Faculty:	Information Technology
Section:	Smart Devices Laboratory

Abstract

As the demand for more intelligent and sophisticated technical devices is increasing all the time, there is a need to provide new and more powerful tools to develop them. The Agora architecture is an attempt to do just that. It builds on the insight that as embedded systems are getting more and more complicated they must be divided into smaller manageable pieces, only exchanging small amounts of data with each other. In this paper I will present some of the theories behind distributed and concurrent object oriented programming as well as development of embedded devices. I will also present the Agora software architecture and its realization using the Smalltalk environment Squeak. Squeak is a modern version of the old Smalltalk system, which showed the way for object oriented programming. Squeak proved to be useful for this project and had many benefits compared with other, newer systems and programming languages. A framework for development of distributed and concurrent embedded systems was created in Squeak with very good results. The Agora system was successfully implemented in the embedded microcomputer BlueMod. This did not only show that Squeak could easily be moved to new hardware architectures, but also the problems with shrinking Squeak to a minimal size. To realize a complete working Agora architecture much work remains, especially with hardware development, but this lies outside the scope of this project. However, successful work has been done with the software implementation of the system which shows the benefits of a system like Agora for development of embedded devices.

Contents

1 Introduction

In this paper the word Smalltalk and Squeak can be read interchangeable. Squeak is an implementation of a Smalltalk system.

The field of embedded devices are advancing in an all-higher speed and they exist not only in our mobile phones but in almost every thing around us, from lamps and refrigerators to toys and calendars. As the demand for more and better embedded systems increases, the need for better tools to develop those increases as well. Current technologies for embedded development are both primitive and ineffective. Compared to development of software on desktop computers you are often fumbling around in the dark when developing in an embedded environment. Little feedback and control over the system is given, and errors are easily created and hard to find and correct. To make it easier for the developer to create more advanced programs, higher-level languages have been invented which handles many of the difficult parts automatically. Unfortunately these technologies have gotten little ground in the embedded area, usually because they are slower and more power consuming. But the computing power of microprocessors is increasing all the time and the power to use better tools is now available.

One technique that has won large ground and acceptance in computer science is Object Oriented (OO) programming which now slowly is gaining ground in embedded programming. OO programming aids the developer in creation of code in both a practical and a mental way and tries to solve the problem with organization of large programs. Now widely used, but unfortunately not as widely understood, OO programming builds on the concept that everything can be built up from simple recursive building blocks. Much like the cell that all living things is built upon, which combined with other and different types of itself, can give form to the most incredible creations and the amazing concept of life that we all are blessed with. OO programming uses instead of a cell, the concept of computers and an OO program is thus built up from many small computers. Each computer works in its own closure and focuses on its own task, but combined they create a complete program.

This concept of having a program built up of small computers is particularly interesting in the field of embedded devices. Especially when we want to have separate execution engines for different processes, which must be linked together to cooperate. The hardest part of working with embedded devices and low level programming is to get hardware and software to work together. If we could use the ideas of OO programming and map that directly to distributed, embedded devices we could create a wonderful symbiosis between the software and the hardware in a system. We no longer would have to care about hardware at all during the development of embedded software. Systems could be built more adaptable, scalable and more reusable. We would simply be able to develop better systems.

1.1 Short history of Squeak

As a background to Object Oriented (OO) programming and the Squeak Smalltalk system, which is the system used to realize our idea, a short history is appropriate.

The history of Squeak starts with the emergence of object oriented programming and ideas taken from some genius system from the early 60's like Lisp [8], Simula [10] and Sketchpad [22]. The ideas from these systems came together in 1966 with Alan Kay at the University of Utah who saw the possibility to combine these ideas to create large, complex and robust software. The first attempt Kay made to create a OO system was FLEX. It was a completely programmable personal computer based on objects. This was in the late 60's so the idea of a personal computer controlled by only a single person was still radical. Kay wanted the personal computer to be flexible and so easy to use that even a child could program it. Although FLEX did not become a major success it helped to create the idea of an easy and flexible personal computer. In 1970 Kay joined Xerox's new Palo Alto Research Centre (PARC) to lead the Learning Research Group, and it was there Smalltalk first was developed. Smalltalk was the first real OO language in the way we think of OO today. It was also the first system to include different features that we see in today's systems like, bit-mapped displays, overlapping windows, menus, icons, and a mouse-pointing device. Smalltalk evolved through several steps from Smalltalk-71 to Smalltalk-80. It was created as a highly portable system, implemented as a byte-code compiler, which used a virtual machine to compile the byte code to the native machines language. This is a technique used more and more as portability today is an important issue in distributed systems like the World Wide Web.

Many people worked at Xerox PARC on creating the Smalltalk we see today but Dan Ingalls can be especially noticed thanks to his great part in implementing and realizing the ideas of Alan. Adele Goldberg was also involved in the implementation and evolution of Smalltalk and one of the authors of the important book about the Smalltalk-80 implementation [7].

In 1995, Kay, Ingalls and Ted Kaehler worked together at Apple Computer. They still envisioned on a flexible computer following the earlier FLEX, which they found still had not been realised. They thought about using Java, but found it too unstable and inflexible. The old commercial Smalltalks available did not either have the flexibility needed. The solution was to create their own version of Smalltalk and to make it free for everybody to use. They created Squeak which was released on the Internet in September 1996 and within weeks Squeak was ported to several platforms and different systems. Today there is a fairly large group of users and developers using Squeak and the system is evolving all the time [9, 15].

2 Project description

2.1 Context

The Agora Architecture [18, 19] is an old idea and a long running project started by Dr. Joaquin Sitte at the Queensland University of Technology (QUT), Australia. It is inspired by the Transputer architecture [12], which was a genius multiprocessor architecture created during the 80's. The Transputer (transistor - computer) was a generic building block for distributed computer systems. The idea of the Agora system is to create a highly modular and easily manageable system, as the Transputer system, but to also incorporate the strengths of interactive OO programming provided by Smalltalk.

2.1.1 The Transputer

A Transputer is a microcomputer with its own local memory and with point-to-point links for communication with other Transputers. The Transputer architecture had a process structure that made distribution and concurrency intuitive and simple. The Occam language [11] was developed to make it simple to program Transputers. It had the important property of a close mapping between the software and the hardware. This made it easy to write distributed programs for distributed Transputer architectures. Another interesting property of the Occam language was the communication protocol using channels between processes which could use either local virtual links or real physical ones. See figure 1 for further explanations. This made it possible to write programs that easily could be adapted to different topologies of distributed Transputers. It was even possible to test your distributed program locally and when it worked, simply distribute it out to the nodes. The point-to-point link protocol used by the Transputer system was automatically synchronized which made it easy to synchronize Transputers to each other. And in contrast to the widely used bus system where the individual bandwidth decreases with each new user of the bus, a new physical connection was added for each new link, which meant that more connected transputers to the system gave higher bandwidth. Inmos Corporation was the company developing the Transputer architecture. It was a beautiful idea but for various reasons it did not become a commercial success and therefore production ended [13, 14].

2.1.2 Agora architecture

The Agora system, as mentioned above, is partly meant to take the place of Transputer systems, but also to extend on the ideas of the Transputer and create a better system. It incorporates the ideas of the Transputer hardware architecture with small microcomputers using point-to-point communication with all its benefits. The main difference is the use of another software architecture. The process model used by the Transputer for concurrency was insufficient for building modular and extensible software for smart appliances. Instead the ob-

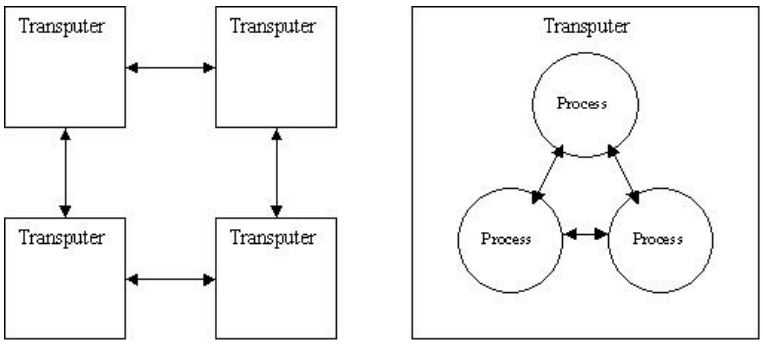

Figure 1: The Transputer uses processes to perform computations. These processes communicate with each other via channels. Channels can exist between processes on the same Transputer and between processes on separate but connected Transputers. This makes it possible to distribute a program in an arbitrary way independent of the physical layout of Transputers.

ject model will be used and the platform used is the Squeak Smalltalk system for reasons discussed at section 2.2.2.

Each processor will run a Smalltalk virtual machine which makes communication between different processors, independent of processor type simple because they all *talk* the Smalltalk language. Communication between processors will completely follow the object model of message passing. This means that processors, actuators and sensors will be regarded as objects and communication with them will not differ from communication with normal objects. The communication will also be synchronized in the same way as the object model Smalltalk uses. The special feature with the Agora object model is that objects are independent of each other and can run in parallel. This means there is no predefined order of computing between different objects, thus making it easier to program with remote objects, which there is no sequential control over. If different objects exist on the same processor they cannot truly run in parallel and the computing power gets shared between them.

The interactive and flexible Smalltalk environment will also be important because in Smalltalk code is added incrementally to running programs. This is possible due to the late binding protocol of Smalltalk. This opens many doors concerning the flexibility for development of systems. It is possible to both code, test and debug a system while it is running, and thus shortening the normal edit-compile-test-debug cycle that is very time consuming and prone to errors. This applies especially for development of embedded devices where interaction with the system usually is very poor.

2.2 Motivations

2.2.1 Why Agora

Embedded systems have usually been small and simple compared to systems on desktop computers and more advanced computers. But the demand for more sophisticated embedded devices is increasing and this is no longer the case. Many technologies have been developed to make devices smarter, usually gathered under the name Artificial Intelligence (AI). Even though we yet are far from realizing the dreams of AI, the complexity of embedded systems are today as high as in other areas of computer science. Classical tools for software development are thus insufficient for advanced development of embedded devices.

A number of crucial differences exist that makes it necessary to create new ways of developing embedded systems.

1. As embedded systems are deployed it interacts with its environment without access by the programmer to see what it perceives. Even during development with current technology it is hard for the programmer to access and to better understand the device's perceptions and reactions of the environment. To make it easier for the developer and possible to understand the program's reactions the development environment needs to be interactive, and development must be possible to do in real time. It must be possible to get information from the system while it is running as well as make changes to it.

2. In embedded systems concurrency of actions is the norm rather than the exception. This is because embedded systems usually interact with the environment, and there is no clear and specified order in how things happen in the real world. The implementation of systems that can handle different tasks in arbitrary order quickly gets very complex, and therewith potential sources of errors. Concurrency is an integral part of embedded systems and effective tools for handling that must be developed.

3. Complicated systems must handle more and more different tasks at the same time, like a robot with lots of different motors and sensors to control and keep track of. Even though it might exist a processor with enough power to handle all the things by itself, the difficulties to write the software for it becomes impossible to realize. Both the concurrency of many different tasks controlled with the same program and the size of these programs becomes major problems. Also the amount of information needing to be sent to and from a single processing unit quickly becomes a problem when the information channels are not quick enough to handle this. In a complicated system it becomes necessary to separate different tasks from each other to create a more manageable system. Separation of duties is a well known concept for solving problems. A real concurrent system with separate embedded processors handling separate tasks becomes more manageable, easier adapts to the concurrency of the real world and man-

ages to handle more information at once. The technology is now available to provide important processes with their own execution engines.

4. Communication between different devices is important for embedded systems. The problem is that there are so many different types of embedded devices using different types of processors and different types of software. This makes it hard to get devices of different types to work together. Special protocols must be developed that specifies a standardised way for the devices to communicate with each other. These protocols are often complicated, hard to implement and still limited in what they can be used for.

2.2.2 Why Squeak

There are a number of reasons to why a Smalltalk system and the Squeak system in particular is chosen to realize this project.

1. The Smalltalk incremental way of developing a system, due to its late binding protocol makes it possible to develop a system while it is running. Smalltalk is also a reflexive language, which means it is able to answer questions about itself. This is a powerful feature which helps the user to interact with a system and get information from it in an effective way. It is possible to be connected to a running device while developing, testing and making corrections. See section 2.2.3.

2. Communication between different types of embedded devices is important when we want to create distributed systems. Smalltalk uses a Virtual Machine (VM) to solve this problem. A VM is a software machine that runs on top of the hardware machine, changing it to a Smalltalk machine. Every machine running the Smalltalk VM is thus a Smalltalk machine that executes Smalltalk code. This is very important because even though there are different types of devices used in a distributed system, they talk the same language and can execute the same code thanks to the VM. This makes it possible to communicate using the existing functionalities of Smalltalk, making special communication protocols unnecessary.

3. Late binding and static typing[1] is debatable and often criticized properties of languages. It is often said that without static typing of a language it is easier to write error prone software. I believe on the contrary that this is wrong in many cases, especially for OO systems. When trying to build large modular systems, late binding is a necessity. Late bindings do not introduce dependencies in the system and the system becomes almost automatically modular. Consideration of how the system interacts, and what types of data to be passed around becomes a local problem. Implementation mistakes from earlier in the process are possible to fix much later on without rewriting the whole system. I also believe that the dynamics created by late bindings makes it possible to create systems capable of more intelligent behaviour. It becomes possible to write dynamic systems suitable for a dynamic environment.

4. Simplicity of programming in Smalltalk is an important issue. Smalltalk is often found to be a bit awkward for experienced programmers, because of its syntactical differences from languages like C and Java. But when getting used to Smalltalk it often proves to be more intuitive and simple than most other languages. The language was developed with ease of use

[1] Most programming languages use different kind of basic building blocks like numbers and characters which is called types. A piece of code usually only works for the specific type it is written for. A statically typed language is type checked at compile time, which is when the code is compiled while a dynamically typed language is type checked during run time, which is when the program is running.

in mind and was aimed for children to use in their school studies. Usually beginners of programming find it easier to work with Smalltalk than with other languages. Why Smalltalk is strange to use for experienced programmers is because Smalltalk lies close to natural languages and is built up of very simple concepts. For example, everything is objects and the only instruction to give is messages to objects. There are no control structures or special commands common to most other languages.

The simple and consistent usage of Smalltalk has proven to be very powerful when building large and complex applications. The ease and intuitive use of the language makes it faster and easier to use for writing applications. Smalltalk is a language that is OO in the important way that it makes it easy and intuitive to write good OO software.

5. To be able to create powerful systems it is necessary to have a powerful tool to create it. Smalltalk gives access to its whole implementation and gives the freedom to change it as one find useful. The available class libraries contain powerful features and give the power to quickly prototype and develop new systems. It is often said that languages like C with pointers and other forms of low level control, gives the developer more freedom and power to create what he or she wants. This is true, however software becomes more difficult to write and it gets terribly easy to create errors. Squeak gives a power that goes far beyond pointers and languages like C, without making things harder.

6. Portability[2] is a very important issue for a system intended to be used in various embedded devices. It must be easy to port the system to different computers and to get them to interact with each other correctly. Squeak is one of the few systems that actually runs bit identical on different machines, and that is important when different Squeak machines is supposed to correctly communicate with each other. Currently Squeak might be the only system running bit identical on more that 30 major platforms, used for everything between advanced VR applications to embedded microcomputer tasks. Without guaranteed bit identical execution on different machines, special protocols for interaction between machines must be specified. This severely limits the communication between devices and makes it harder to implement.

   The ease of porting Squeak is one of the strongest reasons for using Squeak. Squeak is in contrary to many other systems not specified with extensive paper specifications, but generated from itself. Squeak is written in Squeak itself and possible to automatically translate to C code which is easy to compile to new platforms. This makes it possible to create your own Squeak implementation and quickly port it to a new hardware environment.

[2]Porting a system means moving an existing software architecture to a new hardware environment. This is often considered difficult due to the differences in hardware architectures and their demands on the software.

2.2.3 Shortening the Edit-Compile-Test-Debug Cycle

Conventional systems need plenty of time to rebuild a large program after a change. This is particularly painful when working with embedded systems that have to be tested in the real world in real-time. The Smalltalk system uses incremental compilation and dynamic linkage to integrate changes rapidly. Due to dynamic linkage of the system only small local changes need to be made to the system when changes are made. In Smalltalk modules are never statically bound together, instead they are dynamically bound as needed. Dynamic linking is essential to maintain short response time for changing large programs.

Source-level debugging which Smalltalk uses expands on the current debug techniques even though it is almost 30 years old. The usual way to debug in modern languages is that the debugger marks the line where the execution faulted in the source code, and one is able to step through the execution and slowly see what is happening. In Smalltalk the programmer can test a new statement by merely typing it in and view the result while debugging. The System will immediately compile the change and make it able to test. It is possible to continue on a suspended program and correct errors without terminating a program. This gives huge benefits for development of embedded systems [23].

2.3 Objectives

The overall objective of this project is to implement and realize the Agora software architecture [19]. Key parts of the project are:

- Transparency for the user to work with remote and concurrent objects. The distribution of objects must be automatic and the call structure for remote objects and local objects must be identical. This makes it possible to deploy a program on an arbitrary topology of computer nodes and a program gets easy to maintain in a changing environment.
- A dynamic system. Objects must be able to move from different parts of the system while still providing the same access and functionality. It should be possible to create a program locally on a computer, and when it is finished just move the different parts of the program to the desired devices in the system, without any changes in program code.
- Simplicity of the architecture. The implementation will be minimal, only providing the basic functionality. This makes the system easy to use and understand, and also highly adaptable to new ideas and circumstances.
- Consistency of the architecture. The system will make as little changes to the original Smalltalk system as possible to prevent ambiguity and complexity to development.
- Embedded support. The system should be easy to embed in different architectures and an example of this should be done in the BlueMod microcomputer.

3 Related work

The field of concurrent and distributed OO design is not new. It is in fact almost as old as the OO idea itself. Smalltalk is not new either and basically the Smalltalk we use today was created during the 70's. Lately the use of Smalltalk has been overshadowed by languages like C++, Java and the .NET platform and therefore most of the interesting work done with Smalltalk, but surprisingly often in computer science in general, is relatively old. Apart from old research I have been able to relate to some recent interesting projects within the Squeak community.

Some older papers on concurrency in Smalltalk are for example [5] and [25]. Jean-Pierre Briot's paper on generic schedulers for Smalltalk [3] is interesting as well. Another paper by Briot is on Actalk [2], which is a framework for OO concurrent programming and this is actually implemented and available in Squeak.

Interesting work with distributed Smalltalk is for example [1], [4] and [17]. The paper distributed Smalltalk Based on Process-Object model [16] is maybe the paper that most corresponds to this project. This is also the only paper that touches the field of both a distributed and a concurrent system. Some more recent and useful examples are the Magma multi user object database and also Remote Smalltalk (rST) available in Squeak.

Another system worth mentioning is Croquet [20] which is a shared and distributed object environment focused on multimedia and the internet.

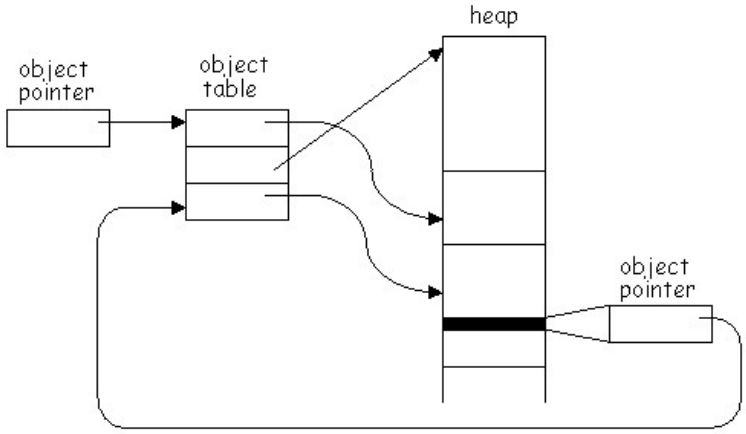

Figure 2: The object table as specified in Smalltalk-80

4 Agora and Smalltalk

The software architecture for Agora imposes two key problems, distribution and concurrency. Both are complex and interesting problems in computer science. They need to be implemented and made working together to make the Agora system work well. Objects must be completely independent of other objects and also independent of location. This means that objects are concurrent and possible to distribute. The goal of this project is to realize this in an intuitive and simple way. The beauty of a system can often be seen by its simplicity.

4.1 Making changes to Smalltalk

To realize this project a study of the Squeak VM was made. The idea was to change the Squeak VM to incorporate concurrent and distributed objects.

The squeak VM consists of the interpreter, which reads and executes bytecode and the object memory, which contains all the objects in the system. The object memory is written in bytecode which is understood by the interpreter. The object memory provides the interpreter with an interface to all the objects in memory that creates the Smalltalk system. Each object is associated with a unique identifier called its object pointer. The object memory and interpreter communicate about objects with object pointers.

In Smalltalk-80 the pointers to objects were redirected via an object table, which made it possible to make programs smaller. See figure 2. This was necessary when Smalltalk was invented, but today size is no longer a major concern and Squeak instead uses direct object pointers pointing directly to objects in memory. This makes the object memory's access to objects faster.

Object pointers are either small integer objects with the most significant (right) bit set to 1, or objects pointers with that bit set to 0. See figure 3

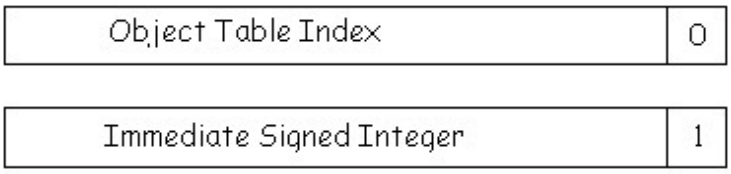

Figure 3: An object pointer is either a pointer to a object in the memory with the most significant bit set to 0. If that bit is set to 1 the object pointer is instead a small integer.

for further explanation. This means that small integers does not get stored as normal objects, but is specified as pointers. This is another feature of Smalltalk to get greater speed [7].

The first idea for creating the Agora system was to change Squeak and have object pointers able to point not only to local objects in the local memory, but also to remote objects. This creates a lot of problems like how to specify remote pointers and remote garbage collection. If Squeak could have an object table the problems would have been easier to solve. The only way to implement this as I can see would be to change the object pointer structure the way small integers does, to specify remote pointers. This will make the number of available local object pointers cut in half and because usually only a small number of remote objects will be used this is a waste of pointers.

The other problem is concurrency in Smalltalk. Smalltalk provides a simple form of concurrency with processes, which can be used to give objects its own execution context. This means that it is possible to run parallel tasks independent of each other. It might be possible to make all Smalltalk objects run concurrently in its own context. Unfortunately this would create a heavy overhead and a vastly bigger execution size of programs, which not is desired in embedded devices where usually the smallest possible computer is used. The functionalities of processes executing in its own context are though available and there might be a possibility to implement this at a core level of Squeak in the future and create a completely concurrent system.

During the investigation of Squeak about how to make changes to the system I came to the conclusion that changing the VM was not the best way to go. First of all, Smalltalk was not at all implemented with concurrency and distribution in mind, and the changes needed to realize this at a core level would become to complex and not suitable for a project of this size. Actually I believe it should be simpler and better to rewrite the system from scratch to realize the ideas. Squeak is already a big and complex system and fundamental changes are not easily incorporated. The other problem is that Squeak proved to be poorly documented, which made it a puzzle to figure out how things worked and how to make the changes.

At this stage the idea to change Smalltalk to incorporate the Agora ideas is still considered, and maybe it will be possible to realize them in the future. One solution could be to leave the VM and object memory as it is, and change the complete class structure to incorporate the ideas of concurrency and distribution. This would be a solution in-between the one to making core changes to the VM and the one presented below which I have done. Further investigation must be made in this area.

4.2 Another path

To make a more manageable project I decided to create a layer on top of Squeak that provided concurrency and distribution of objects. This follows some of the earlier implementations discussed in chapter 3 on concurrent Smalltalk and Distributed Smalltalk. My first idea was to use Actalk and rST, which already was available in Squeak as platforms to work from. Both systems turned out to be too big to be usable considering my goal of having a minimal implementation. They were also difficult to adapt to the ideas I had as well as making them work together in the desired way. This made me decide to implement my own solution from scratch. For concurrency I used the Active Object (AO) theory, also used in Actalk and described in for example [2] and [5]. This creates objects that are tightly coupled with their own process. For distribution I decided to use the Proxy Object (PO) model described in [4] and [17]. This provided a way of referring to all objects, including the remote ones as they were local, thus making a program independent of objects distribution. The POs also turned out to be good to use together with AOs. The PO used for distribution is an integral part of the AO, which makes AOs easy to distribute. Distributed objects running their own process also solves the problem of remote objects being garbage collected by mistake as proposed in [16]. This whole concept was possible to blend together into a seamless and integrated implementation with AOs and POs working together.

4.3 Active Objects

Active Objects comes from the idea of having objects with their own active context. That means they actively execute some computation by them self, instead of simply answering when called upon via messages. For example it is possible to build an AO that can run concurrent asynchronous computation by implementing a message queue as part of the object. Now the object can receive many messages at once even if it is busy executing an old message. The object might also be able to execute several activities at once, and schedule the activities in its own way. An example of such an object is provided in figure 4.

4.4 Proxy Objects for object distribution

Proxy Objects are a simple but powerful way of remote communication with objects in an OO system. The idea is to use a local object that simply forwards

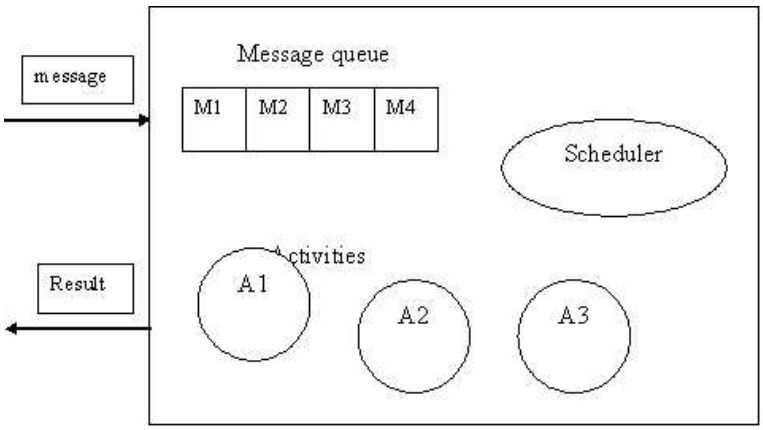

Figure 4: This active object can receive several messages at the same time and stores them in a message queue. The internal scheduler handles the execution of the messages which is called activities. The object can run several activities in parallel.

messages to the remote object and returns the answer. Locally a remote object is thus considered local, because a message is sent locally to the PO, which pretends to be the remote object. This makes communication with remote object completely transparent and also very simple. It becomes possible to move objects to remote devices without making changes to the code. It is even possible to have objects moving around on different nodes in a system while a running program is using the object, and without the program needing to know that the object is mobile. A simple sketch describes the PO in figure 5.

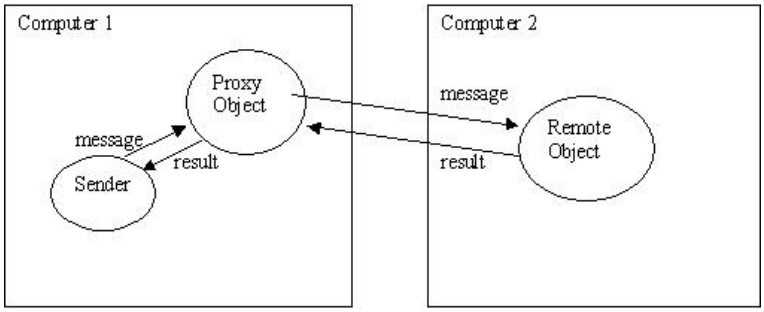

Figure 5: Proxy object. Messages to a remote object are sent to a local proxy object. The proxy object forwards the message to the remote object in the remote computer over a link. The result is sent back the same way.

4.5 The Agora software implementation

To realize the Agora software architecture AOs were used for concurrency and POs for communication. As a preliminary distribution protocol the Internet TCP[3] protocol was used to communicate between different running Squeak systems. This made it easy to run several Squeak systems on the same computer, as well as on separate computers connected to a network. The communication protocol is built to be easily adapted to new circumstances, so the TCP protocol can easily be changed to a new one.

4.5.1 Concurrency

The core activity in the Agora AO model is to let each object run its own process. This is implemented by using the simplest AO possible. The Agora AO has a process that waits for a message, executes it and then returns the answer when it is ready. The PO communicates with the process via virtual links which synchronizes the communication. This may sound suspiciously similar to normal objects, and in some way it is because there is still synchronous communication and the object is limited to executing one message at a time. The important difference is that the execution of the message is done by its own process, independent from other objects. This automatically makes each AO concurrent and possible to run in parallel with other AOs. The importance of running objects in parallel lies in the possibility to build concurrent systems able to react on a dynamic environment. One AO waiting for a message will not block other AOs that might need to respond before the waiting AO.

The AO used in the Agora system is called AgoraObject, and it is used as the base class for all AOs in an implementation. The only difference in using AgoraObjects compared to normal objects is that an AgoraObject is chosen as

[3]Transfer Control Protocol, manages data communication over distributed architectures like the Internet

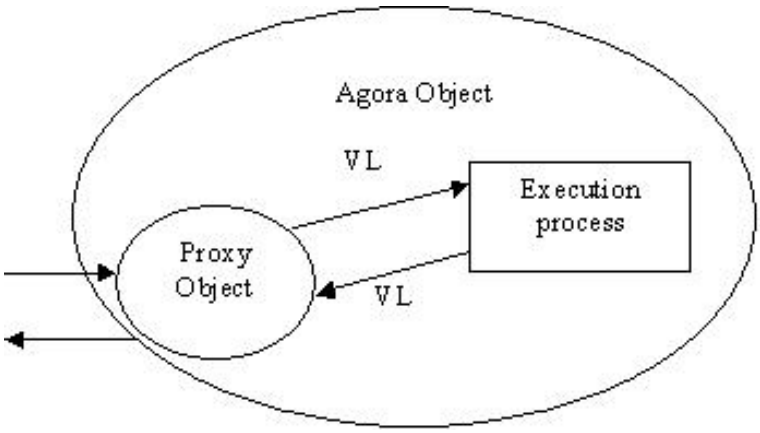

Figure 6: AgoraObject. Messages are sent to a proxy object inside the AgoraObject and forwarded via a virtual link (VL) to an execution process. The execution process performs the message, independent of other processes in the system and returns the message via the internal proxy object.

the base class. All the functionalities are then inherited from the AgoraObject to all new objects which also become AgoraObjects. This is a very intuitive way of using AOs following the basic inheritance scheme of OO programming. The only thing the programmer needs to keep track of is what base class to use when implementing AOs. Figure 6 explains the internal structure of AgoraObjects.

4.5.2 Time sharing

In a concurrent architecture time sharing is an important issue. If different concurrent processes are distributed on different devices they are possible to run at the same time in parallel. This is quite natural, but the problem comes when we want to simulate this real parallelism on the same device with a single processor. One processor can only handle one specific task at one time. To create a shared environment for the different processes, the processor time is then shared between the processes in a certain way. This is also called time slicing because the available processor-time gets sliced up in pieces and divided between the processes. This can be done in a number of ways, but the simplest way is to give an equal amount of run-time to each process by simply executing each process after the other for a short time over and over again. This simulates a true parallel behaviour which we need to create a concurrent system. In Squeak there is no time sharing between processes. They run independent from each other but each process runs until it is finished. In a time shared system, the process runs for a little while, say 1 ms and then lets the next process continue. This is implemented in Smalltalk/X [21] for example, and can relatively easily be implemented in Squeak as well. Processes in Squeak can be sent the *wait*

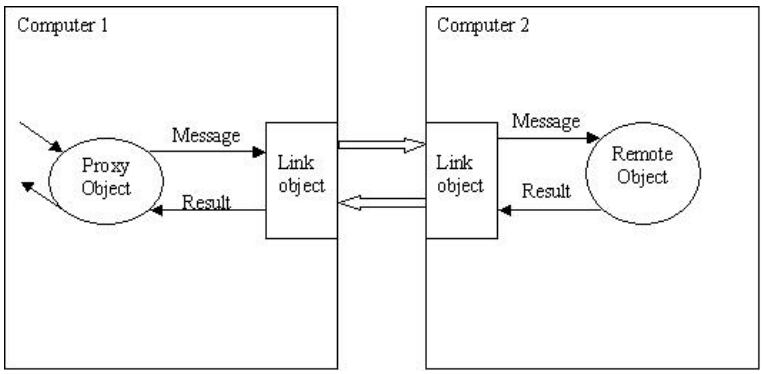

Figure 7: The distribution protocol. The link-object transfers messages and works as a layer on top of the real link between the computers. The proxy object passes received messages to the link object. The link-object transfers the message to the remote link object where messages are forwarded to the remote object. The result is passed back in the same way.

message that forces the process to halt and let the next process run. It is easy to implement a scheduler in Squeak that keeps track of the objects that is supposed to run in parallel and force them to time slice. For various reasons that I will explain later, this has not been implemented in the Agora implementation. This is basically due to the uncertainty of what Squeak platform to use, but should be easy to implement in the desired way when that is decided. An example implementation of time sharing for Smalltalk-80 can be seen in the paper *A Concurret Prototype in Smalltalk-80* [5].

4.5.3 Distribution

The distribution is implemented by using a simple proxy model. The PO in the Agora system is an object that is connected to a link-object, which handles the communication with remote systems. This link-object is a generic object easily adapted to new types of architectures and link types. The PO has the property of forwarding each message it receives to the link-object it is connected to. It also waits for the result when that is computed in the remote object which makes the communication synchronized. When the PO receives the computed result from the remote object, it returns that result to the caller. Figure 7 explains the functionality of the AgoraProxy object and distributed communication.

4.5.4 Communication

The Agora system sends objects to communicate, and because Smalltalk is a pure[4] OO language, everything can be sent to communicate. There are no limitations of what to be communicated and distributed between different machines. The most common object to be sent is the message object to command a remote object do a computation. But except of messages it is possible to create remote classes, send local objects over to the remote machine and communicate directly with the remote compiler.

4.5.5 Security

In a distributed system security immediately becomes an issue. If the system is accessible from the outside like with the TCP protocol on the current Agora implementation it is of importance to prohibit unauthorized use. Even if the system is not accessible from the outside, security is important for protection from error prone software. It is possible to remotely control everything in the Agora system, but that is usually not desired and even dangerous. To incorporate the OO idea of encapsulation which is important for modular and extensible software it must be possible to specify what kind of messages is accepted at certain nodes of a distributed system. In the Agora architecture one can specify what kind of messages a system accepts. This also makes every Agora system possible to regard as an object and encapsulation of that object handles the security.

[4]When a object oriented language is defined pure it means that everything in the system or in the language is objects. There are no special types or structures that not are objects.

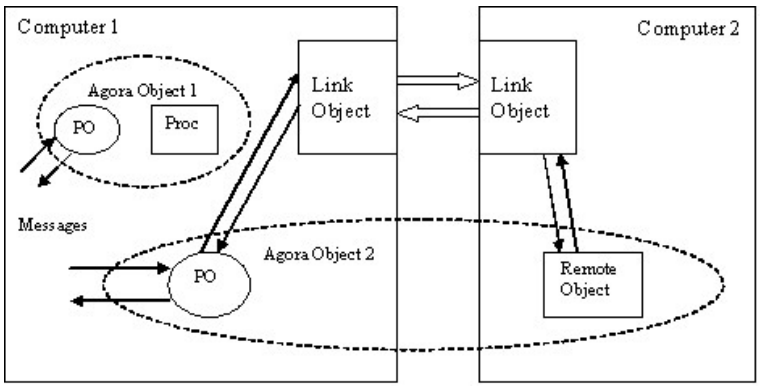

Figure 8: This shows the complete structure of the Agora system. AgoraObjects can be both local and distributed, but messages are sent in the same way for both cases.

4.5.6 Architecture

The complete structure of the Agora implementation is showed in figure 8. It shows how the PO plays a crucial role for both the AO which are local on the computer and to the objects that are remote. Transparency of the distribution is handled by the PO model. Messages to a remote object are sent in the same way as messages to local objects, because a PO is a local object.

4.5.7 Agora classes

The most relevant classes in the Agora system are presented below.

1. **AgoraObject**
   This is the base class to create AOs. Any object supposed to be active should inherit from this class. When the AgoraObject is created the reference to its internal proxy (AgoraORB) is returned.

2. **AgoraORB**
   This is the PO in the Agora system. ORB stands for Object Request Broker which is another term used for POs. Actalk uses this term and AgoraORB's name is inspired from there.

3. **AgoraVirtualLink**
   This object handles synchronous communication between objects. For example when AgoraORB sends a message to the execution process this message goes through an AgoraVirtualLink. Because the AgoraVirtualLink is synchronized this automatically handles synchronization of the system.

4. **AgoraLinkManager**
   The link manager is the link to the outside of Squeak. The AgoraLinkMan-

ager object is connected to another AgoraLinkManager object on another node in a distributed system. Messages to a remote object go through these link objects. The AgoraLinkManager specifies what types of messages that are possible to send to it.

5. **AgoraMessage**
   This is the message object that is sent between two AgoraLinkManager objects over a communication link.

6. **AgoraLinkProtocol**
   This class is created as a description over how to implement platform specific link protocols. It can be seen as an abstract class and used as an implementation example.

7. **AgoraTCPProtocol**
   This class is an implementation of the TCP protocol for the Agora architecture. The AgoraLinkManager uses this class to send data with the TCP protocol. This class makes it possible to communicate with other Agora computers over the internet, or communicate between different Squeak instances on the same computer.

5 Embedded Devices

The main target for the Agora system is development of embedded devices and systems using embedded devices. Even though the Agora system has been implemented on a PC and can be used on personal computers, the real benefits of the system comes when used for development of embedded devices.

First of all, the interactive development environment provided with Squeak brings a huge benefit compared to current systems. Especially the ability to use incremental programming while being connected to the device is something that would make big improvements. Another issue is that today's embedded systems more and more moves toward distributed collaborative architectures. It is too complex to build a robot for example with humanlike legs that is being controlled by a single computer. The computation power must be distributed to several smaller computational entities, exchanging only necessary information. The computer deciding where to go will only have to tell the leg computer to move the leg forward, not instruct every single motor in the leg and ask every sensor what is happening. As with all systems, abstraction is important when the system becomes large and complex.

5.1 Porting the system

The ability to port the system to various embedded devices is essential for the Agora architecture. The desired scenario is different types of devices being able to work together and understand each other. It is here the benefits of a VM like the Squeak VM really shows. The VM transforms each machine it runs on to a Squeak machine, talking the same language. This makes it much easier to build a distributed system when no special communication protocol is needed. Instead it is possible to use normal Squeak messages to talk to other machines. No consideration is needed for communication between different types of machines because they all are Squeak machines.

The Agora system runs on the Squeak system and because of this the porting of the Agora system basically means porting Squeak. One nice feature with Squeak is that it is written completely in Squeak itself (actually the Slang language is used, which is a subset of Smalltalk that is easy to translate to C code). Any changes needed to the system are possible to do inside Squeak. Squeak contains a Smalltalk to C translator which makes it possible to translate the whole system, or parts of it to generic C code that easily can be compiled to new systems. The C code generated from Squeak is very modular and simple, and contains only simple standard-C code. Porting Squeak to a new system should be unproblematic as long as there is a C compiler for the system. The only parts needed to be implemented are the platform dependent parts like input devices or graphical output on a display. On embedded devices these features are usually both simple and few. Therefore the porting of Squeak is a relatively simple and straightforward process.

A very good feature with Squeak being generated to standard C code is that Squeak is easily ported to new hardware systems that not have an oper-

ative system running on it. Implementing software on a bare system can be a very complex process, but Squeak makes this very simple when most hardware architectures comes with a C compiler.

5.2 Shrinking the image

Smalltalk basically consists of two parts, the VM and the image. The image is the compiled bytecode which is executed by the VM to create Squeak. It contains the entire execution state and all objects in the system. When saving Squeak one actually takes a copy from the computer's memory and writes that to the image file. This makes it possible to save an exact state of the system and restart from that exact state. This means that there is no start-up sequence when starting Squeak. Squeak simply start from the same state it was at during the last save. The possibility of saving an exact state is very effective but makes it harder to make certain changes to the system.

To port Squeak to an embedded device, the size of the system is very important. Usually the embedded device of choice is as small and cheap as possible for the desired task. A cheaper device means a more attractive product at the market and a small product can make the product more usable and less power consuming, thus longer time of use if run on batteries. The Squeak system is intended to run on a desktop computer with a pointing device and a graphical screen. The Graphical User Interface (GUI) is an important and highly integrated part of the system. With good reasons that is, because not many systems can provide a better and more interactive GUI. The GUI is unfortunately a big problem for an embedded implementation of Squeak. It takes a lot of space and computing power and is usually not at all needed for embedded devices.

No good image for embedded devices was available during the time course of this project. Some attempts of embedded Squeak versions had been made, but usually the GUI was still left in the image taking up much unnecessary space. An implementation aimed for embedded support called Squeakette [6] was found, which proved to be easy to work with. Unfortunately the GUI was still left in that image. The image was also from an old version of Squeak, so this was not the ideal product to use. Squeakette proved though to be a good start and test ground to try things with.

The differences in the old Squeakette image and the newer versions of Squeak created some problems for the implementation of the Agora system. The time sharing part is the most obvious which I decided to not implement because of this. Most of my development was done in a new Squeak version, Squeak 3.4 and this code had some trouble moving to Squeakette. It might have been possible to implement time sharing on both versions, but because I saw Squeakette only as a first step and not the final version to use, I decided to not implement the time sharing. This part should though be easy to implement when a decision has been made on what Squeak implementation to use for the Agora architecture.

5.2.1 Better image on its way

During the time of my project, work within the Squeak community was aimed at making Squeak more suitable for embedded products. The two major tasks were to clean up the system and to shrink the image. Currently there are many dependencies in the system, especially to the GUI. Therefore shrinking the image is not done easily and the Squeak community wanted to really open the doors for Squeak to conquer the embedded world as well. A major cleanup of the entire system was undertaken as well as an effort to shrink the image. This seems promising for the future but unfortunately this work did not get finished in time for my project. Hence I had to try to shrink an image myself.

Because there is no certain start-up procedure of a Smalltalk image and there were many dependencies in the system, shrinking the image proved to be a difficult task. Within the Squeak community, image engineering is often considered a black art or some sort of witchery instead of a science. I must agree to those statements and shrinking the image felt like doing an operation blindfolded with a sharp knife. If the wrong things were taken away in the wrong way, the system would just stop working. It was very hard to get rid of all the dependencies that were in the system. I succeeded however with stripping down a new image (Squeak version 3.5) to a stable state without the GUI to the size of 500Kb. I was also able to create a less stable image with a size under 400Kb. With some more work put in to this, it should be possible to make a much better and smaller image than the one I worked on. But during my project there was no point in putting too much work on this, when the Squeak community was working on a Squeak version better suitable for embedded applications.

5.3 The BlueMod

The embedded device that was used for the port was the BlueMod [24] made by HCV Wireless. This product was mainly chosen due to availability but also because it is a good platform to test remote communication with. The BlueMod is a Bluetooth embedded microcontroller aimed at providing simple implementations of Bluetooth systems. This is very interesting because the Agora system could be used to make distributed computations over Bluetooth, which would be a nice test example of the system. The BlueMod can also be used with other Bluetooth devices running Squeak and the Agora software to communicate and execute distributed Agora programs. For example many PDAs (Personal Digital Assistant) have Bluetooth functionality, and as Squeak already is ported to many of them this would be a interesting usage of the system.

6 Evaluation

The aim of this project was to implement the Agora software architecture in the Squeak Smalltalk environment and to port the system to an embedded device.

These parts have both successfully been accomplished. An investigation on changing the entire Squeak system including the VM was made but was omitted due to the complexity of the problem. The implementation of the Agora software system was instead done as a layer on top of Squeak. The system was implemented using the TCP communication protocol and tested in a distributed environment. It worked extremely well as test implementations of distributed and concurrent programs were easily created using it. The system was also fast, providing a very small overhead of only 20% when heavily used. For most usage the overhead was close to zero. Unfortunately we did not have time to do a real distributed embedded test of the system using different BlueMods communicating with each other. But shortly after my project was finished a simple test of this was accomplished in a successful manner and more work on this would be done further on.

A port of the system to an embedded device was made, and the device used was a Bluetooth device called BlueMod. The porting was successful and proved that it is possible and even simple to port the system to new devices. The ported system was made using an old version of Squeak because of its simplicity and its small size. A newer version of Squeak would have been better to use and would have created a better Agora system. A new Squeak version would be compatible with current Squeak releases and incorporate all the technical advantages the new Squeak releases incorporates. An example of the benefit of this would be to use the full Squeak system as the development environment for the embedded system. A better, smaller and more modular Squeak system was being worked on within the Squeak community and was available shortly after this project was finished. Thus, the use of a better Squeak version for the Agora system should not be far off.

7 Future work

1. The first part to be worked on should be to get the Bluetooth communication working. It would be nice to use the BlueMod to communicate with other Bluetooth devices and the Agora architecture would make distributed bluetooth applications easy to create. This would make it possible to test the Agora software architecture in a real embedded environment. At this moment only personal computers have been used to test the Agora system in a distributed environment.

2. The Squeak version used for the Agora project should be changed to a new Squeak version. The Squeak version 3.6 which was worked on during the time of my project will bring many benefits. It will be easier to port the system to new hardware environments and compatibility with current Squeak versions would be achieved. Squeak 3.6 is also cleaned up from many dependencies in the system and thus easier to shrink and embed.

3. Time scheduling should be implemented in the new Squeak version. This is necessary to create a true shared environment so that a process that takes long time to evaluate not blocks other processes.

4. A development environment for embedded Squeak implementations should be made. This would create easy access and development of remote images. A simple browser that can display the remote images classes and send changes would be a great tool.

5. Further development of the Agora architecture is needed. Especially the hardware part of the system needs more work. It would be truly great to have actuators and sensors talking native Smalltalk and sending and receiving Agora messages. As a start it is possible to use existing systems, but to get native Agora devices, implemented in for example FPGAs[5] would be truly great. It would be easy to access these devices with the Agora architecture to create distributed systems.

[5]FPGA - Field Programmable Gateway Array is a programmable hardware circuit. It is possible to control the FPGA's logic gates from software and thus create a custom hardware device.

8 Conclusions

In this paper I have described the theory behind distributed and concurrent object oriented programming for embedded environments and the software implementation of the Agora system following these theories. The Agora system provides an important simplification for development of smart appliances. Not only does it provide an object oriented environment for developers, but it also maps the environment on top of the hardware in a way that make objects change from mental simplifications of a problem to physical parts of the system. The system used to implement this was the Squeak Smalltalk system, which is one of the few systems that could make it possible to realize our ides. Squeak provides an incremental way of programming which is very helpful during development of embedded devices. An implementation of the Agora software system proved to be most suitable to implement on top of the Squeak system. An intentionally small and simple system was created that provided concurrency and distribution. The Agora system works exceptionally well when tested on computers, and nearly no overhead is caused by using it. The system puts an abstraction layer over distribution, concurrency, synchronization and communication, which usually are difficult areas to handle. A port of the system has also been made to the embedded BlueMod device. This shows how easy it will be to move the system to different platforms and to get them to communicate with each other.

Even though I think the project has gone well there is more work to be done to make this a complete system. For the software part there needs to be a better development environment implemented and a better image is needed that is smaller and up to date with the current Squeak implementations. The most important part though is the development and use of new hardware created for the Agora architecture.

9 Acknowledgements

I want to thank...

Prof. Joaquin Sitte for giving me the chance to take part in this project from where I have learnt a great deal.

Smart Devices Laboratory for the support and recourses I have been given.

Anna Lindberg for her support all along my project and for her important help with the final corrections.

Marc Slater for the help with the BlueMod.

Umeå University and Per Lindström for the support from my home university.

References

[1] John K Bennet. *The Design And Implementation of Distributed Smalltalk.* Technical report, Department of Computer Science, University of Washington, 1987.

[2] Jean-Pierre Briot. *Actalk: A Framework for Object-Oriented Concurrent Programming - Design and Experience.* Technical report, Laboratoire d'Informatique de Paris 6, 1999.

[3] Jean-Pierre Briot and Lic Lescaudron. *Design of a Generic and Reusable Scheduler for Smalltalk-80.* Technical report, LITP, Institute Blaise Pascal, 1989.

[4] A Corradi, L Zannini, and M Leonardi. *Distributed Environments based on Objects: upgrading Smalltalk toward distribution.* Technical report, Dipartimento di Elettronica, Univisersita'di Bologna, 1990.

[5] Antonio Corradi and Leonardi Letizia. *A Concurret Prototype in Smalltalk-80.* Technical report, Dipartimento di Elettronica, Univisersita'di Bologna, 1990.

[6] Kurtz Fernhout. *Kurtz-Fernhout software.* 1999. Available from: http://www.kurtz-fernhout.com/squeak/ [Accessed 20 August 2003].

[7] A Goldberg and D Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[8] Paul Graham. *Lisp.* 2001. Available from: http://paulgraham.com/lisp.html [Accessed 20 August 2003].

[9] Mark Guzdial. *Squeak: Object-Oriented Design with Multimedia Applications.* Prentice Hall, 2001.

[10] Jan Rune Holmevik. *The history of Simula.* University of Trondheim, 1995. Available from: http://java.sun.com/people/jag/SimulaHistory.html [Accessed 20 August 2003].

[11] Inmos. *OCCAM Programming Manual.* Prentice-Hall Internationall, 1984.

[12] Inmos. *Transputer Reference manual and Product Data.* Prentice Hall, 1985.

[13] Inmos. *Transputer Instruction Set - A compiler writer's guide.* Prentice Hall, 1988.

[14] Inmos. *Transputer Technical Notes.* Prentice Hall, 1989.

[15] Alan Kay. The early history of Smalltalk. *ACM SIGPLAN Notices, Volume 28 No. 3*, 1993.

[16] Y S Lee, J H Huang, and F J Wang. *A Distributed Smalltalk Based on Process-Object Model.* Technical report, Telecommunication Laboratories and national Chiao-Tung University, Taiwan, 1991.

[17] Paul L McCullough. *Transparent Forwarding: First Steps.* Technical report, Xerox PARC Northwest, 1987.

[18] Joaquin Sitte. *Distributed Multiprocessor Computing for Smart Appliances.* Technical report, Smart Devices Laboratory, QUT Australia, 2001.

[19] Joaquin Sitte. *A Replacement Transputer Node for the Agora Distributed Computing Architecture for Robotics.* Technical report, Smart Devices Laboratory, QUT Australia, 2002.

[20] David A Smith, Andreas Raab, David Reed, and Alan Kay. *Crouqet.* Technical report, Viewpoint Research Institute, 2002.

[21] Except software AG. *Smalltalk/X.* Except software AG, 2003.
Available from:
http://www.exept.de/sites/exept/english/Smalltalk/frame_uebersicht.html
[Accessed 20 August 2003].

[22] Ivan E Sutherland. *Sketchpad, a man-machine graphical communication system.* MIT Libraries, 1963. Available from:
http://theses.mit.edu/Dienst/UI/2.0/Describe/0018.mit.theses/1963-10
[Accessed 20 August 2003].

[23] David M Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* Massachusetts Institute of Technology, 1987.

[24] HCV Wireless. *HCV Wireless mainpage.* 2003.
Available from: http://www.hcv.com.au [Accessed 20 August 2003].

[25] Yokote Yasuhiko and Tokoro Mario. *The Design and Implementation of ConcurrentSmalltalk.* Technical report, Department of Elactrical Engineering, Keio University, Japan, 1986.